НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Факультет інформатики та обчислювальної техніки

Кафедра автоматики та управління в технічних системах

До захисту допущено:

Завідувач кафедри

_____ Олександр РОЛІК

«___»_____20__ р.

Дипломний проєкт

на здобуття ступеня бакалавра
за освітньо-професійною програмою «Програмне забезпечення інформаційно-комунікаційних систем»
спеціальності 121 «Інженерія програмного забезпечення»
на тему: «Протокол асинхронного обміну повідомленнями на основі UDP»

Виконав (-ла):
студент (-ка) IV курсу, групи
Лесогорський Кирило Сергійович _____

Керівник:
Ассистент кафедри АУТС,
Дорога-Іванюк Олена Олександрівна _____

Рецензент:
доцент кафедри ПЗКС ФПМ, к.т.н. ,
Цуркан Василь Васильович _____

Засвідчую, що у цьому дипломному проєкті немає запозичень з праць інших авторів без відповідних посилань.
Студент (-ка) _____

Київ – 2020

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Факультет інформатики та обчислювальної техніки
Кафедра автоматики та управління в технічних системах

Рівень вищої освіти – перший (бакалаврський)

Спеціальність – 121 «Інженерія програмного забезпечення»

Освітньо-професійна програма «Програмне забезпечення інформаційно-комунікаційних систем»

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Олександр РОЛІК
«___»_____20__ р.

ЗАВДАННЯ

на дипломний проєкт студенту
Лесогорському Кирилу Сергійовичу

1. Тема проекту «Протокол асинхроного обміну повідомленнями на основі UDP», керівник проєкту Дорога-Іванюк О. О. асистент кафедри АУТС, затверджені наказом по університету від «7» травня 2020 р. №1081-с

2. Термін подання студентом проекту: «9» червня 2020 р

3. Вихідні дані до проекту: операційна система Ubuntu 16.04.6, мова програмування Java, серидовище програмування Intellij IDEA Community Edition, цільова платформа Java

4. Зміст пояснювальної записки: постановка задачі, вибір та обґрунтування технологій, специфікація протоколу, архітектура програмних компонентів, тестування програмного забезпечення.

5. Перелік графічного матеріалу: схема життєвого циклу протоколу, схема пакетів протоколу, схема компонентів брокеру, схема компонентів клієнту

6. Дата видачі завдання «3» березня 2020 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів виконання дипломного проєкту	Термін виконання етапів проєкту	Примітка
1.	Вивчення рекомендованої літератури	10.04.2020	
2.	Аналіз існуючих рішень	13.04.2020	
3.	Розроблення специфікації протоколу	20.04.2020	
4.	Розроблення транспортного шару протоколу	24.04.2020	
5.	Розроблення архітектуру клієнту та брокеру	26.04.2020	
6.	Реалізація програмної частини системи та подальше тестування	29.04.2020	
7.	Тестування програми	14.05.2020	
8.	Виконання графічних документів	17.05.2020	
9.	Оформлення пояснювальної записки	18.05.2020	
10.	Подання ДП на попередній захист	25.05.2020	
11.	Подання ДП рецензенту	08.05.2020	
12.	Подання ДП на основний захист	15.05.2020	

Студент Кирило ЛЕСОГОРСЬКИЙ

Керівник проєкту Олена ДОРОГА-ІВАНЮК

АНОТАЦІЯ

Лесогорський К.С. Протокол для асинхронного обміну повідомленнями на основі UDP. КПІ ім. Ігоря Сікорського, Київ, 2020.

Ключові слова: протокол, UDP, обмін повідомленнями.

Основна частина документу викладена у пояснювальній записці, виконаній на 70 сторінках, складається з 5 розділів, містить 18 рисунків, 15 посилань на джерела та 4 кресленики.

Об’єктом дослідження дипломного проєкту є протокол асинхронного обміну повідомленнями на основі UDP.

У першому розділі сформульовано вимоги до програмного рішення та протоколу та проаналізовано існуючі рішення. У другому розділі розглянуто та проаналізовано технології, які можна використати при розробці програмного забезпечення. У третьому розділі було сформульовано специфікацію протоколу, описано основні механізми, необхідні для його роботи. У четвертому розділі було описано архітектуру та реалізацію брокера та клієнта. У п’ятому розділі було описано техніки контролю та верифікації працездатності програмного забезпечення.

Клієнт та брокер реалізовані з використанням високорівневої мови програмування Java. Для досягнення максимальної швидкодії використовуються неблокуючі методи вводу-виводу. Панель керування брокером реалізована з використанням MVC фреймворку Spring.

SUMMARY

Lesohorskyi K. S. UDP-based asynchronous messaging protocol. Igor Sikorsky KPI, Kyiv, 2020.

Keywords: protocol, UDP, messaging.

The bulk of the document is set out in an explanatory note, made on 70 pages, consists of 5 sections and contains 18 figures, 15 references to sources and 4 drawings.

The object of research of the diploma project is the protocol of asynchronous exchange of messages on the basis of UDP.

The first section formulates the requirements for the software solution and protocol and analyzes the existing solutions. The second section discusses and analyzes technologies that can be used in software development. In the third section the specification of the protocol was formulated, the basic mechanisms necessary for its work were described. The fourth section described the architecture and implementation of the broker and the client. The fifth section describes the techniques for monitoring and verifying the functionality of the software.

The client and broker are implemented using a high-level Java programming language. Non-blocking I / O methods are used to achieve maximum performance. The broker control panel is implemented using the Spring MVC framework.

Поз.	Формат	Позначення	Найменування	Кількість аркушів	№ прим.	Примітки
1			Документація загальна			
2						
3			Знову розроблена			
4						
5	А4	ІТ61.110БАК.005 ПЗ	Протокол асинхроного обміну	70		
6			повідомленнями на основі UDP			
7			Пояснювальна записка			
8	А3	ІТ61.110БАК.005 Д1	Протокол асинхроного обміну	1		
9			повідомленнями на основі UDP			
4			Життєвий цикл протоколу			
5	А3	ІТ61.110БАК.005 Д2	Протокол асинхроного обміну	1		
6			повідомленнями на основі UDP			
7			Схема пакетів протоколу			
8	А3	ІТ61.110БАК.005 Д3	Протокол асинхроного обміну повідомленнями	1		
9			на основі UDP			
10			Архітектура брокеру			
11	А3	ІТ61.110БАК.005 Д4	Протокол асинхроного обміну	1		
12			повідомленнями на основі UDP			
13			Архітектура клієнта			
14						
15						
16						
17						

					ІТ61.110БАК.004 ТП		
Змн.	*Лист*	*№ докум.*	*Підпис*	*Дата*			
Розроб.		Лесогорський К.С.			Протокол асинхронного обміну повідомленнями на основі UDP. Відомість технічного проєкту	*Літ.*	*Лист* 1 / *Листів* 1
Перевір.		Дорога-Іванюк О.О					
Н. Контр.						*КПІ ім. Ігоря Сікорського кафедра АУТС гр. ІТ-61*	
Затверд.		Ролік О.І.					

Пояснювальна записка
до дипломного проєкту
на тему: «Протокол асинхронного обміну повідомленнями на основі UDP»

Київ – 2020

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

Брокер черг – серверне програмне забезпечення, яке приймає, обробляє, зберігає та маршрутизує повідомлення, які до нього надходять.

Консьюмер — споживач повідомлень. Також відомий як підписник у моделі публікувальник-підписник.

Продюсер — генератор повідомлень. Також відомий як публікувальник у моделі публікувальник-підписник.

AMQP – Asynchronous Message Queue Protocol. Сучасний протокол брокеру черг, який базується на TCP, але підтримує і альтернативні транспортні протоколи.

Congestion control – механізм, який запобігає перевантаження мережі через постійні спроби перевідправити повідомлення у випадку збою.

DRY — dont repeat yourself, принцип написання коду, при якому досягається максимальне перевикористання кода через такі механізми, як композиція та наслідування.

Packet Jittering – скупчення пакетів у вузькому місці мережі, при якому мережевий пристрій перевантажується, що призводить до затримок та втрати пакетів.

RTT – Round Trip Time. Час, за який пакет надходить від джерела до отримувача і назад.

SOLID — п’ять принципів написання коду, що роблять його легше у підтримці та доробці. До цих принципів відносять принцип єдиної відповідальності, принцип закритості до зміни, відкритості до розширення, принцип підстанови Барбари Лісков, принцип сегрегації інтерфейсів та принцип інверсії залежностей.

TCP – Transmition Control Protocol. Протокол транспортного рівня, який використовується для гарантованої доставки повідомлень мережею Інтернет.

UDP – User Datagram Protocol. Один з найпростіших протоколів транспортного рівня, який використовується для обміну повідомленнями у мережі Інтернет. Не гарантує доставку повідомлень.

ВСТУП

Технології для обміну повідомленнями активно використовуються в сучасних розподілених обчислювальних системах. Їх використання надає необхідний транспортний рівень для реалізації шаблону публікувальник-підписник і дає змогу відділити отримувачів від відправників, що дає змогу підвищити надійність і знизити зв’язність системи.

На сьогоднішній день існує багато реалізацій протоколів для асинхронного обміну повідомленнями. Вони надають різні API та можливість та мають свої переваги та недоліки. Але їх поєднує одна особливість — у якості транспортного рівня вони використовують протокол TCP. Цей протокол гарантує доставку, але і додає досить високі накладні витрати. З іншого боку протокол UDP, який є високоефективним протоколом негарантованої доставки повідомлень дуже не активно використовується у цій сфері.

Велика кількість додатків базується на обробці великих потоків даних, які майже неможливо обробити на одному фізичному сервері. Такі сфери, як транслювання подій наживо, інтернет речей, аналітичні продукти та багато інших генерують тисячі подій у секунду. Обробку подібних потоків даних доводиться розподілювати між декількома серверами, що значно підвищує складність та збільшує накладні витрати на синхронізацію та розв’язання конфліктів. Тому створення протоколу асинхронного обміну повідомленнями та реалізації необхідної інфраструктури для його роботи – актуальна задача, яка зможе знизити складність та підвищити швидкодію існуючих та нових систем.

Метою проєкту є створення специфікації прототипу протоколу асинхронного обміну повідомленнями, який буде використовувати у якості транспортного рівня протокол UDP та реалізація інфраструктури, необхідної для його роботи, а саме брокер (серверна частина) та клієнт (клієнтська частина протоколу).

Предметом та об’єктом дослідження є вимоги та необхідні компоненти для реалізації подібного протоколу. Так, основним завданням протоколу є

передача інформації між продюсерами та консьюмерами. Передача відбувається через проміжковий брокер, який зберігає повідомлення, мультиплексує та демультиплексує повідомлення, балансує навантаження. Важливо подбати про захист інформації в транзиті, можливість адміністрування додатку та складність розгортання та підтримування у робочому стані отриманого комплексу програмного забезпечення.

Для досягнення поставленої мети необхідно:

— проаналізувати наявні рішення, виявити їх переваги та недоліки;

— проаналізувати технології, використання яких надасть змогу створити програмне забезпечення, що задовольняє вимогам;

— створити специфікацію протоколу, у якій врахувати шифрування, гарантованість доставки, механізми контролю навантаження мережі;

— створити програмне забезпечення брокеру та клієнту, що задовольняють вимогам та реалізують специфікацію протоколу.

З практичної точку зору таке програмне забезпечення дозволить підвищити ефективність існуючих рішень, які працюють з високими об’ємами даних та знизити кількість обчислювальних ресурсів, необхідну на обробку. Також таке програмне забезпечення дозволить знизити витрати на супроводження, які є великою частиною витрат, оскільки підтримка вузлів розподілених вузлів є досить ресурсоємною операцією.

З наукової точку зору створення подібного протоколу є досить цікавим, оскільки використання UDP у останні роки починає зростати, навіть у протоколах, де необхідна гарантована доставка. Так, Google розробляє протокол QUIC – протокол гарантованої доставки, який використовує UDP у якості протоколу транспортного рівня. Цей експеримент виявився досить успішним, тому специфікація протоколу HTTP/3 також використовує UDP у якості транспортного протоколу[1]. Це пов'язано з моральною застарілістю TCP протоколу та складності його модифікації, оскільки це вимагає оновлення великої кількості програмного забезпечення, що фізично неможливо. Ця проблема не виникає для UDP протоколу, оскільки він не реалізує майже ніяких

механізмів доставки повідомлень і дає змогу реалізувати їх у просторі користувача. Дослідження можливості використання UDP у протоколах асинхронного обміну повідомленнями може дати дуже позитивні результати та підкреслити важливість подальшого розвитку протоколів на основі UDP.

1 ПОСТАНОВКА ЗАДАЧІ

1.1 Функціональні вимоги

Функціональні вимоги визначають основний фронт робіт розробника, регламентують поведінку застосунку в тій чи іншій ситуації. Вони ставлять завдання, які повинна виконувати система

Основною метою цього проєкту є створення прототипу протоколу для асинхронного обміну повідомленнями на основі UDP. Необхідно дослідити, які проблеми мають наявні протоколи та спробувати розв'язати їх за допомогою протоколу UDP.

Оскільки сам по собі протокол не має сенсу, необхідно також створити брокер повідомлень та клієнт, які дозволили б провести навантажувальне тестування та зрозуміти які переваги та недоліки має цей протокол.

1.1.1 Вимоги до брокеру

Брокер повідомлень повинен мати можливість створювати та видаляти черги та партиції. Черга це абстракція потоку повідомлень. Саме у них продюсери надсилать повідомлення. До черги під’єднується декілька партицій, з яких можуть читати консьюмери.

Брокер має впроваджувати API для створення, керування та видалення черг та партицій. При створені черги вказується наступна інформація: назва та кількість приєднаних партицій. Після створення топіку до нього можна додавати або видаляти додаткові партицій. Видалення партицій це потенційно небезпечна операція через ризик втратити дані з черги, тому чергу не можна видалити при наявності в ній повідомлень, або при наявності під’єднаних до неї консьюмерів. Але існують ситуації, коли видалення переповненої або не відповідаючої партиції може запобігти більш критичним втратам. На такий випадок існує механізм force операцій, які можна виконати лише з привілеями адміністратора системи.

Брокер має бути кросплатформним, тому бажано використання мов, які використовують віртуальні машини, такі як Erlang(BEAM), сімейство JVM мов, або мови, що підтримують .NET Core.

1.1.2 Вимоги до клієнту

Клієнт має впроваджувати інтерфейс для взаємодії з брокером у двох режимах: продюсер та консьюмер. В обох режимах режимах роботи клієнт використовуватиме асинхронне API. При створенні продюсеру клієнт має вказати логін, пароль та партицію, до якої будуть надсилатись повідомлення. При створенні консьюмеру клієнт вказує логін, пароль та топік, з якого будуть отримуватись повідомлення.

Клієнт відповідальний за інформування брокеру про обробку повідомлення.

Клієнт має бути реалізованим на тій самій мові, що і брокер та використовувати виключно асинхронний API.

1.1.3 Вимоги до протоколу

Протокол має бути незалежним від імплементацій та базуватись на UDP. Також протокол має впроваджувати базові механізми безпеки, такі як користувачі та шифрування.

Користувачі мають два рівні привілеїв: адміністратор і користувач. При запиті на створення сесії необхідно передати інформацію про автентифікацію, а при успішній автентифікації дані про користувача зберігаються у сесії.

Для шифрування має використовуватись симетричний алгоритм шифрування з генерацією ключів при створені сесії.

1.2 Нефункціональні вимоги до програмного продукту

Нефункціональні вимоги, визначають вимоги до системи в цілому, а не в окремих випадках використання, описують властивості і характеристики, які повинна демонструвати система, а також обмеження, які повинні бути дотримані в окремих випадках. Виділено такі нефункціональні вимоги:

— кросплатформність ― клієнтська бібліотека та брокер повинні бути незалежні від архітектури пристрою та операційної системи;

— висока пропускна здатність ― одна з основних переваг; використання UDP можливість значно підвищити пропускну здатність, що важливо у певних випадках використання протоколу;

— легкість розширення ― оскільки протокол розробляється як прототип, при створені клієнту і брокеру необхідно врахувати складність модифікації для створення другої версії протоколу;

— експлуатаційна придатність ― брокер та клієнт повинні бути придатні до використання у незалежності від часу;

— конфіденційність ― протокол повинен надавати базове шифрування для безпечної передачі та цілісності повідомлень.

1.3 Огляд існуючих рішень

Системи обміну повідомлень дуже активно використовуються у багатьох доменах, де необхідна обробка даних близька до реального часу, наприклад фінансові системи, стримінгові платформи, інтернет речей. Тому аналогів розробляємого рішення існує багато — RabbitMq, Apache Kafka, ZeroMq, NATS і багато інших. Здебільшого ці системи мають подальшу спеціалізацію.

1.3.1 Apache Kafka

Розподілений журнал Apache Kafka спеціалізується на гарантованій потоковій обробці, має гарні здібності до кластеризації, що дуже важливо для розподілених систем, але вимагає значно більших ресурсів для запуску, налаштування, роботи та підтримки[2]. Основними абстракціями в Kafka є журнал(топік), який складається з набору партицій. Загальну схему роботи Kafka зображено на рисунку 1.1

Краще за все Kafka можна характеризувати як розподілений та реплікований журнал фіксації змін. Його основні властивості:

— розподілений, оскільки Kafka розгортається як кластер вузлів, як для стійкості до помилок, так і для масштабування;

— реплікований, оскільки повідомлення зазвичай реплікуються на декількох вузлах (серверах);

— журнал фіксації змін, тому що повідомлення зберігаються в сегментованих, append-only журналах, які називаються топіки. Ця концепція журналювання є основною унікальною перевагою Kafka.

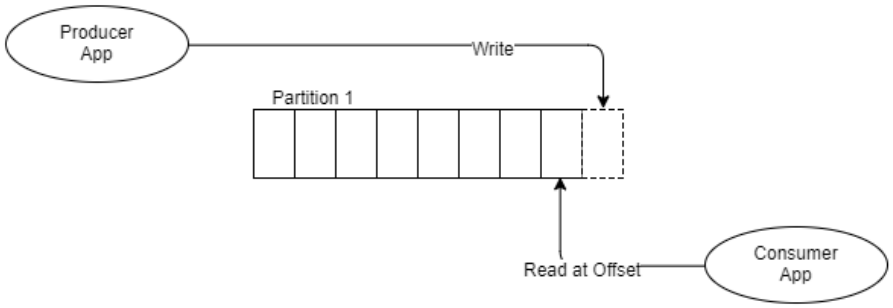

Рисунок 1.1 — Схема роботи Kafka

Замість того, щоб поміщати повідомлення в чергу FIFO і відстежувати статус цього повідомлення в черзі, Kafka просто додає його в журнал.

Повідомлення залишається, незалежно від того, чи буде воно отримано один або кілька разів. Видаляється воно відповідно до політики утримування даних (retention policy, також званий window time period).

Кожен одержувач відстежує, де вона знаходиться в журналі: є покажчик на останній отримане повідомлення і цей покажчик називається адресою зміщення. Одержувачі підтримують цю адресу через клієнтські бібліотеки, і в залежності від версії Kafka адреса зберігається або в ZooKeeper, або в самій Kafka.

Відмітна особливість моделі журналювання полягає в тому, що вона миттєво усуває безліч складнощів, що стосуються стану доставки повідомлень і, що більш важливо для одержувачів, дозволяє їм перемотувати час назад, повертатися і отримувати повідомлення за попередньою відносною адресою.

Кожна партиція представляється окремим файлом, в якому гарантується черговість повідомлень. Порядок повідомлень гарантується тільки в одній партиції. Надалі це може привести до деякого протиріччя між потребами в черговості повідомлень і потребами в продуктивності, оскільки продуктивність в Kafka масштабується кількістю партицій. Партиція не може підтримувати конкуруючи одержувачів.

Повідомлення можуть бути перенаправлені до партицій за циклічним алгоритмом або через функцію хешування.

Резюмуючи, можна сказати що Kafka досить складне та комплексне рішеня, яке має свої переваги та недоліки.

Серед переваг основними є:

— легкість масштабування;

— гарантія чергової доставки та обробки;

— доставка повідомлень “рівно один раз”;

— можливість відмотки повідомлень.

Головні недоліки:

— високі вимоги до інфраструктури;

— складність розгортання та підтримки;

— більші вимоги і складність клієнту.

1.3.2 RabbitMQ

RabbitMQ - це розподілена система управління чергою повідомлень. Розподілена, оскільки зазвичай працює як кластер вузлів, де черги розподіляються по вузлах і, за потребою, реплікуються в цілях стійкості до помилок і високої доступності[3]. Штатно, вона реалізує AMQP 0.9.1 і пропонує інші протоколи, такі як STOMP, MQTT і HTTP через додаткові модулі.

RabbitMQ використовує як класичний, так і новаторський підходи до обміну повідомленнями(див. рисунок 1.2). Класичний в тому сенсі, що вона орієнтована на чергу повідомлень, а новаторський - в можливості гнучкої маршрутизації. Саме ця можливість маршрутизації є її унікальною перевагою. Створення швидкої, масштабованої і надійної розподіленої системи повідомлень само по собі є досягненням, але функціональність маршрутизації повідомлень робить її справді визначною серед безлічі технологій обміну повідомленнями.

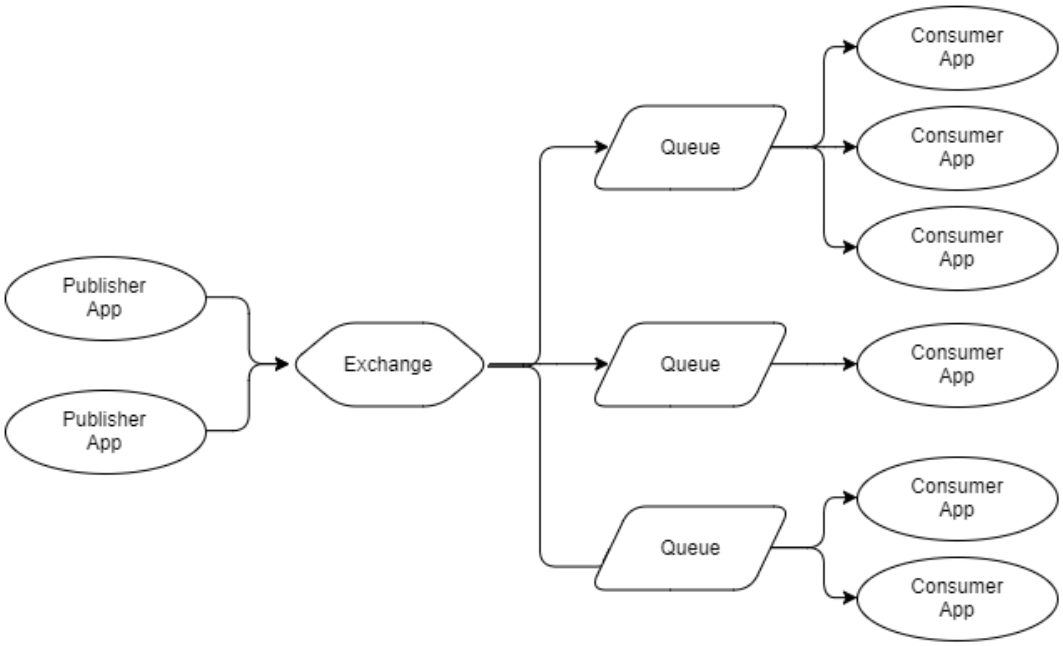

Рисунок 1.2 — Схема роботи Rabbit MQ

Загальну послідовність роботи RabbitMq можна описати наступним чином:

— паблішери (publishers) відправляють повідомлення на точки обміну(exchange'i);

— точки обміну(exchange'i) відправляють повідомлення в черги та в інші точки обміну;

— RabbitMQ відправляє підтвердження паблішеру при отриманні повідомлення;

— одержувачі (consumers) підтримують постійні TCP-з'єднання з RabbitMQ і оголошують, яку чергу або черги вони прослуховують;

— RabbitMQ проштовхує (push) повідомлення одержувачам;

— одержувачі відправляють підтвердження успіху / помилки;

— після успішного отримання, повідомлення видаляються з черг.

RabbitMQ дає гарантії «одноразової доставки» і «хоча б однієї доставки», але не «рівно однієї доставки».

Повідомлення доставляються в порядку їх прибуття в чергу (в кінці кінців, це і є визначення черги). Це не гарантує, що завершення обробки повідомлень збігається з тим же самим порядком, коли у вас є конкуруючи одержувачі. Це не помилка RabbitMQ, а фундаментальна реальність паралельної обробки упорядкованого набору повідомлень. Цю проблему можна вирішити, використовуючи Consistent Hashing Exchange.

Exchange'i - це маршрутизатори повідомлень для черг і / або інших exchange'ей. Щоб повідомлення переміщалася з exchange'а в чергу або на інший exchange, необхідна прив'язка. Різні exchange'i вимагають різних прив'язок. Існує чотири типи exchange'ей і пов'язані з ними прив'язки:

— fanout (розгалуження). Приєднує в усі черги і обмінники, які мають прив'язку до exchange'у. Стандартна модель публікувальник-підписник.

— direct (прямий). Маршрутизує повідомлення на основі ключа маршрутизації, який несе з собою повідомлення. Ключ маршрутизації задається

публікатором. Ключ маршрутизації - короткий рядок. Прямі обмінники відправляють повідомлення в черги / exchange'і, у яких є ключ сполучення, який точно відповідає ключу маршрутизації.

— topic (тематичний). Маршрутизує повідомлення на основі ключа маршрутизації, але дозволяє використовувати неповну відповідність;

— header (заголовки). RabbitMQ дозволяє додавати до повідомлень заголовки одержувачів. Заголовки exchange'і передають повідомлення відповідно до цих значеннями заголовка. Кожна прив'язка включає в себе точну відповідність значень заголовка. До прив'язці можна додати кілька значень з ЯКИМИ або УСІМА значеннями, необхідними для відповідності.

— consistent hashing (консистентне хешування). Це обмінник, який хешує або ключ маршрутизації, або заголовок повідомлення, і відправляє тільки в одну чергу. Це корисно, коли вам потрібно дотримати гарантії порядку обробки та при цьому мати можливість масштабувати одержувачів.

Загалом серед переваг RabbitMQ можна виділити:

— легкість розгортання і підтримки;

— легкість створення консьюмерів;

— широкий функціонал для маршрутизації повідомлень.

До недоліків:

— нижча(у порівнянні з Kafka) пропускна здатність;

— відсутність механізму доставки рівно один раз;

— відсутність можливості «перемотки часу».

1.4 Огляд механізмів гарантованої доставки у протоколі TCP

TCP – протокол транспортного рівня моделі OSI. Його було розроблено у 1974 році. З того часу протокол зазнав декілька змін, але основні концепти залишились незмінними[4]. На відміну від UDP, TCP забезпечує надійну доставку сегментів, послідовність сегментів при отримуванні, роботу з сесіями та контроль за швидкістю передачі.

Під надійною доставкою розуміють автоматичне повторне пересилання не отриманих сегментів. Кожен сегмент маркується за допомогою спеціального поля - порядкового номера (sequence number). Після відправки певної кількості сегментів, TCP відправник очікує підтвердження від одержувача, в якому вказується порядковий номер наступного сегмента, який адресат бажає отримати. У разі, якщо таке підтвердження не отримане, відправка автоматично повторюється. Після деякої кількості невдалих спроб, TCP вважає, що адресат не доступний, і сесія розривається.

Таким чином, надійна доставка означає, що розробник ПЗ, який використовує TCP на транспортному рівні знає, що якщо сесія не розірвалася, то все що він доручив відправити буде доставлено одержувачу без втрат. З цієї причини багато протоколів рівня додатків використовують для транспорту TCP.

Кожен сегмент на нижній рівнях TCP / IP обробляється індивідуально. Тобто, як мінімум, він буде запакований в індивідуальний пакет. Пакети йдуть по мережі і проміжні маршрутизатори в загальному випадку не знають про те, що запаковано в ці пакети. Часто пакети з метою балансування навантаження можуть йти по мережі різними шляхами, через різні проміжні пристрої, з різною швидкістю. Таким чином одержувач може отримати сегменти не в тому порядку, в якому вони відправлялися.

TCP автоматично перезбере їх в потрібному порядку використовуючи поле порядкових номерів і передасть після склейки на рівень додатку.

Контроль за швидкістю передачі дозволяє корегувати швидкість відправки даних в залежності від можливостей одержувача.

Завдяки механізму плавучого вікна (sliding window), TCP може працювати з мережами різної надійності. Механізм плавучого вікна дозволяє змінювати кількість пересланих байтів, на які треба отримувати підтвердження від адресата. Чим більше розмір вікна, тим більший обсяг інформації буде переданий до отримання підтвердження. Для надійних мереж підтвердження можна надсилати рідко, для зменшення трафіку, тому розмір вікна в таких мережах автоматично збільшується. Якщо ж дані губляться, розмір вікна

автоматично зменшується. Таким чином, для ненадійних мереж, розмір вікна повинен бути мінімальним.

Механізм плавучого вікна дозволяє TCP постійно міняти розмір вікна - збільшувати його поки все нормально і зменшувати, коли сегменти не доходять. Таким чином, в будь-який момент часу розмір вікна буде більш-менш відповідати стану мережі.

Структура TCP графічно зображена на рисунку 1.3. Розглянемо структуру більш детально:

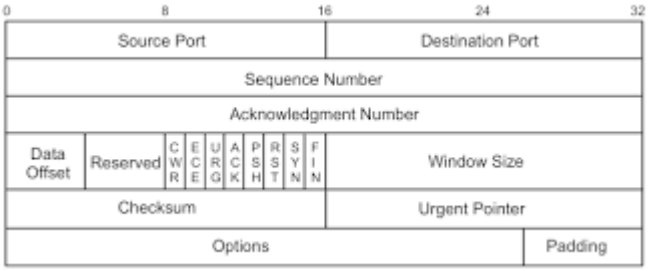

Рисунок 1.3 – Загальна схема TCP пакету

— source port і destination port ─ це відповідно номера портів відправника і одержувача, що ідентифікують додатків на відправляє і приймає вузлах;

— sequence number і acknowledgment number ─ це порядковий номер сегмента і номер підтвердження, які використовуються для надійної доставки;

— data offset ─ Це чотирьох бітне поле, що містить в собі довжину заголовка TCP сегмента;

— reserved ─ 6 зарезервованих біт;

— control ─ поле з прапорами, які використовуються в процесі обміну інформацією та описують додаткове призначення сегмента. Наприклад, прапор FIN використовується для завершення з'єднань, SYN і ACK ─ для установки;

— window size ─ містить розмір вікна;

— checksumm ─ контрольна сума заголовка і даних;

— urgent pointer ─ ознака важливості (терміновості) даного сегмента;

— options ─ додаткове необов'язкове поле, яке може використовуватися, наприклад, для тестування протоколу;

— у розділі даних містяться власне дані, отримані від протоколу рівня додатків, або їх шматок, якщо дані довелося розбивати.

Висновки до розділу

Під час проведення аналізу вимог до програмного забезпечення було розглянуто та проаналізовано як наявні аналоги брокерів черг, так і механізму досягнення гарантованої доставки протоколу TCP. При розгляданні існуючих рішень було виявлено недоліки та переваги кожного з розглянутих рішень, щоб досягнути компромісу який дасть велику кількість контролю розробникам клієнтських застосунків, але при цьому буде зручним у використанні і практичним навіть на невеликих проєктах.

2 ВИБІР ТА ОБҐРУНТУВАННЯ ТЕХНОЛОГІЙ

2.1 Цільова платформа брокеру та клієнту

Клієнт та брокер мають бути кросплатформні та не залежати від архітектури пристрою. Основною цільовою операційною системою є Linux, можливе використання асинхронного IO API epoll з подальшою можливістю додавати драйвера для інших операційних систем, які використовують інші API для асинхронного IO, наприклад kque або IO completition port. Як було зазначено у функціональних вимогах, важливою буде підтримка таких операційних систем:

— Windows;

— Linux;

— FreeBSD;

— MacOS.

2.2 Вибір віртуальної машини

Віртуальні машини активно використовуються з 60 років минулого сторіччя. Вони пропонують багато переваг у порівнянні з класичною схемою AOT компіляції. При використанні віртуальної машини вихідні коди компілюються у байт коди віртуальної машини, а під час виконання віртуальна машина перетворює байт-код у машинні команди машини виконавця. Це дає змогу компілювати код один раз і виконувати його на різних машинах з етапом JIT-компіляції.

Основними причинами популярності віртуальних машин є платформонезалежність, низькі накладні витрати у порівняні з інтерпретаторами та легкість написання.

Тому для забезпечення платформонезалежністю клієнту та брокеру доцільним буде використання мови, що виконується на віртуальній машині, але

при цьому мають низькі накладні витрати. Серед таких віртуальних машин особливо виділяються JVM, .NET Core та BEAM.

JVM – це перевірена часом та дуже поширена віртуальна машина. Існує декілька реалізацій JVM, кожна з яких має свої переваги та недоліки. Перемикатись між реалізаціями JVM можна не змінюючи вихідний код. Багато мов можуть компілюватись у Java байт код. Серед них основними є Java, Scala, Kotlin та Clojure. Мови JVM сімейства сумісні один з одною, окрім Clojure(існують певні обмеження при виклику Clojure коду з інших мов через особливість динамічної типізації).

.NET Core – відносно молода віртуальна машина. Платформонезалежна версія .NET Framework, що пропонує низькі накладні витрати та широкий спектр підтримуваних мов. Активно використовується у корпоративному програмному забезпеченні. Підтримує великий перелік мов, таких як F#, C# та Visual Basic. Містить багато оптимізацій та можливостей для написання низькорівневого коду, включаючи покажчики та аллокацію пам’яті на стеку, а у загальному випадку пропонує оптимізований GC для автоматичного керування пам’яттю на кучі. У інфраструктурних додатках ця платформа не знайшла популярності через те, що довгий проміжок часу була доступна лише для операційних систем сімейства Windows.

BEAM - Bogdan's/Björn's Erlang Abstract Machine. Віртуальна машина для сімейства мов Earlang. Це сімейство включає в себе такі мови як Earlang та Elixir. Популярна для телекомунікацій та OTP. Забезпечує високу надійність, відмовостійкість та легкість розгортання, але додає більше накладних витрат у порівнянні з іншими віртуальними машинами. Має невелику спільноту через специфічність цільових мов.

Врахувавши всі ці фактори, в якості віртуальної машини було обрано JVM, через такі фактори:

— низькі накладні витрати;

— перевірена часом;

— використовувалась для створення систем подібного класу(Kafka);

— широкий спектр підтримуваних мов;

— високопродуктивний збірник сміття(garbage collector).

2.3 Вибір мови програмування

Мова програмування також є важливим вибором. Вона значно вплине на стиль, складність підтримки та якість написання програми. Оскільки в якості віртуальної машини було обрано JVM, то вибір мов зводиться до досить вузького кола, а саме Scala, Java та Kotlin.

Scala — це сучасна мульти-парадигмальна мова програмування, розроблена для вираження загальних концепцій програмування в простій, зручній і типобезопечній манері. Скала поєднує краще від світу об’єктно орієнтованого та функціонального програмування, пропонуючи унікальну гібридну парадигму.

Scala — це об'єктно-орієнтована мова в тому сенсі, що кожне значення — це об'єкт. Тип і поведінка об'єктів описані в класах і трейтах (характеристиках об'єктів). Класи розширюються за рахунок механізму наслідування і гнучкого змішування класів, яке використовується для реалізації множинного наслідування.

Scala також є функціональною мовою в тому сенсі, що кожна функція — це значення. Scala надає легкий синтаксис для визначення анонімних функцій, підтримує функції вищого порядку, підтримує вкладені функції, а також каррінг. Scala має вбудовану підтримку алгебраїчних типів даних, які використовуються в більшості функціональних мовах програмування (ця підтримка базується на механізмі зіставлення зі зразком, де в якості зразку виступають класи-зразки). Об'єкти надають зручний спосіб угруповання функцій, що не входять в клас.

Java — високорівнева об’єктно орієнтована мова програмування. В останніх версіях в цю мову почали впроваджувати часткову підтримку функціонального програмування[5].

Java відносно стара і стабільна мова програмування, що на відміну від Scala пропонує підтримку бінарної сумісності між версіями мови.

Java активно використовується для побудови інфраструктурних додатків та складних корпоративних систем, що свідчить про її достатню надійність та високу продуктивність.

Велика кількість бібліотек написана на Java та велика кількість спеціалістів використовує Java в якості основної мови програмування, тому розвивати продукти написані на Java легше, ніж на будь-якій іншій мові.

Kotlin — молода гібридна мова програмування. Kotlin розроблявся як мова, що дуже близька до Java, але при цьому розв'язує багато проблем Java і значно зручніша у використанні. Kotlin розв'язує такі проблеми Java як null-safety, імперативність, делегування, розширення класів, корутини, тощо. Kotlin значно простіший за Scala, але розв'язує подібні проблеми у трохи іншому світлі

Ці переваги дозволили Kotlin дуже швидко стати популярним у мобільній та десктоп розробці, але у серверній розробці він не набув популярності.

Враховуючи всі фактори було обрано мову програмування Java. Керівними факторами у цьому рішенні була бінарна сумісність та активне використання Java у інших продуктах. Хоч протокол і буде прототипом, у разі успіху планується подальший його розвиток за моделлю open source. Тому бінарна сумісність грає велику роль у майбутньому розвитку протоколу.

Окрім того, Kafka спочатку розроблялась саме на Scala, але пізніше мігрувала на Java через проблеми бінарної сумісності.

2.4 Вибір транспортного протоколу

В якості транспортного протоколу було обрано UDP, як протокол з низькими накладними витратами, що дає змогу реалізувати більшість необхідних алгоритмів на рівні додатку.

User Datagram Protocol (UDP) — протокол транспортного шару. UDP є частиною пакету Інтернет-протоколів UDP / IP. На відміну від TCP, це ненадійний та безсесійний протокол[6]. Отже, немає необхідності встановлювати з'єднання до передачі даних.

Хоча протокол управління передачею (TCP) є основним протоколом транспортного рівня, який використовується в більшості Інтернет-служб, забезпечує надійну доставку, надійність та багато іншого, але всі ці послуги коштують нам додаткових витрат та затримок. Для таких послуг у режимі реального часу, як комп’ютерна гра, голосовий або відеозв'язок, конференції в реальному часі нам потрібен UDP. Оскільки потрібна висока продуктивність, UDP дозволяє скидати пакети замість обробки затримок пакетів. У UDP немає перевірки помилок, тому це також економить пропускну здатність.

Протокол User Datagram (UDP) є більш ефективним з точки зору затримки та пропускної здатності.

UDP пакет складається з заголовку і даних. Схему UDP пакету зображено на рисунку 2.4

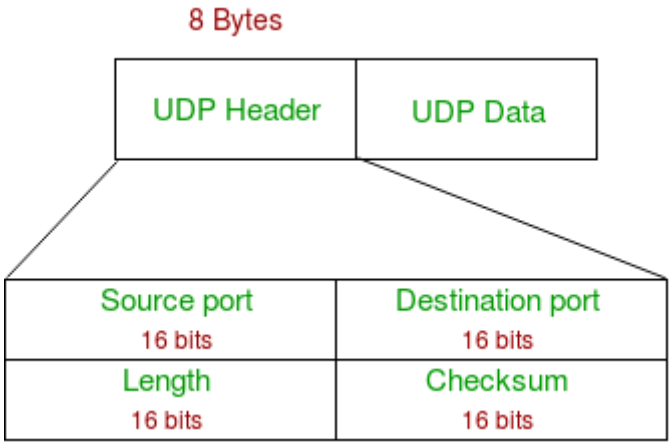

Рисунок 2.4 — Схема UDP пакету

Розглянемо цю схему детальніше:

— source port(порт джерела) — це 2-байтне поле, яке використовується для ідентифікації номера порту джерела;

— destination port(порт призначення) ― Це 2-байтне поле, яке використовується для ідентифікації порту призначеного пакету.

— length(Довжина) ― довжина UDP, включаючи заголовок та дані. Це 16-бітове поле.

— контрольна сума ― це 2-байтне поле. Це 16-бітове доповнення суми доповнення заголовка UDP, псевдо-заголовка інформації із заголовка IP та даних, доповнених нульовими октетами в кінці (якщо потрібно), зробити октети кратні двом.

UDP використовується для простого зв'язку з відповіддю на запит, коли розмір даних менший, а отже, є менша стурбованість щодо контролю потоку та помилок. Він підходить для протоколів багатоадресної передачі, оскільки UDP підтримує комутацію пакетів. UDP використовується для деяких протоколів оновлення маршрутизації, таких як RIP (протокол інформації про маршрутизацію).

Зазвичай UDP використовується для додатків у режимі реального часу, які не переносять нерівномірні затримки між розділами отриманого повідомлення. Наступні протоколи прикладного рівня використовують UDP як протокол транспортного рівня:

— NTP (мережевий протокол часу);

— DNS (служба доменних імен);

— BOOTP, DHCP;

— NNP (Network News Protocol);

— Протокол цитати дня;

— TFTP, RTSP, RIP, OSPF;

— Часова мітка.

UDP приймає датаграму з мережевого шару, додає її заголовок і відправляє користувачеві. Отже, він працює швидше, аніж TCP/IP .

2.5 Вибір фреймворку для панелі керування

Панель керування буде використовуватись для адміністративних та конфігураційних задач. Це значно спростить керування брокером та дозволить отримувати додаткову інформацію про стан системи у майже реальному часі. Оскільки панель керування не є основним компонентом системи, вона має бути не вибагливою до ресурсів системи. Для створення панелі керування буде використано простий та мінімалістичний веб-інтерфейс, тому веб-сервер буде вбудовано до брокеру, а взаємодія між ними буде організована з використанням моделі акторів, що значно знизить зв’язність та підвищить надійність обох компонентів. На сьогоднішній день у світі Java існують декілька популярних фреймворків для веб-застосунків. Особливо перспективними є Spring та Play

Spring — дуже популярний фреймворк для розробки веб-застосунків. Він розробляється за моделлю open-source. Він включає в себе багато бібліотек і розв'язує типові проблеми, з якими стикаються програмісти при розробці веб-додатків. Основною перевагою використання Spring є простота та низькі додаткові витрати. Spring підтримує як MVC модель додатку, так і створення виключно API додатків. Spring пропонує програмістам інверсію управління(Inversion of Control, IoC) з використанням ін’єкції залежностей(Dependency Injection, DI)[7]. Це дуже добре накладається на модель POJO. Також Spring має інтеграції з багатьма технологіями, що у майбутньому допоможе расширювати функціонал додатків з легкістю.

Play — відносно молодий і інноваційний Java та Scala фреймворк. Він, як і Spring, пропонує рішення типових проблем при розробці веб-додатків, але більше концентрується на асинхронному вводі/виводі на відміну від Spring, який більше концентрується на можливостях самого фреймворку. Play використовує Akka для асинхронного вводу-виводу. Це дуже потужна бібліотека для роботи з потоками даних, використання котрої значно полегшить розробку та розвиток самого брокеру. Детальніше цю бібліотеку буде розглянуто у розділі 2.6.

Резюмуючи, можна сказати, що Play — досить непоганий вибір для панелі керування, але його можливості асинхронного вводу-виводу будуть мати дуже малий вплив на продуктивність додатку, тому використання Spring MVC з його широким спектром можливостей більш доцільне у даній ситуації.

2.6 Асинхронний ввід-вивід

2.6.1 Моделі багатозадачності

Оскільки розробляємe програмне рішення дуже сильно зав’язано на мережевий ввід/вивід, то його оптимізація є дуже важливим елементом досягнення високої продуктивності брокеру. Використання традиційного синхронного блокуючого підходу з використанням декількох неконтрольованих потоків виявився дуже неефективним з експоненційною вимогою до ресурсів(проблема 10 000 під’єднань)[8].

Проблема полягає у великих накладних витратах створення та підтримки потоків на рівні операційної системи, оскільки кожен потік резервує пам’ять під стек та використовує обчислювальний час процесору. Окрім того перемикання процесів на рівні операційної системи — дуже дорога операція, при якій поточний стан потоку(стан регістрів) зберігається у оперативну пам’ять і планувальник операційної системи обирає який потік має виконуватись наступним. Цей підхід відомий як переважна багатозадачність(preemptive multitasking).

Для розв’язання цієї задачі використовують різні шаблони кооперативної багатозадачності(cooperative multitasking). При цьому підході в середині програмного процесу створюється невеликий(часто за кількістю обчислювальних ядер) пул фізичних потоків. В середині процесу існує планувальник, який відповідає за розподілення використання фізичних потоків між віртуальними потоками, також відомими як “зелені” потоки(green threads). Ці потоки майже не несуть накладних витрат.

Але одним з найголовніших аспектів кооперативної багатозадачності є механізм розподілу фізичних потоків між віртуальними. При даній моделі планувальник “довіряє” потокам, тому виділяє їм стільки процесорного часу, скільки вони потребують (рисунок 2.5). Коли віртуальний потік закінчує свою роботу, або є заблокованим в очікуванні певного ресурсу — він віддає контроль планувальнику, який вирішує кому віддати вільний потік. Таким чином накладні витрати щодо адміністрування потоками зводяться до мінімуму і процесорний час

утилізується максимально ефективно.

Використання кооперативної багатозадачності требує змін у тому, як програмісти пишуть код. Оскільки при використанні цієї форми багатозадачності функції можуть переривати своє виконання не дійшовши до кінця, то необхідно

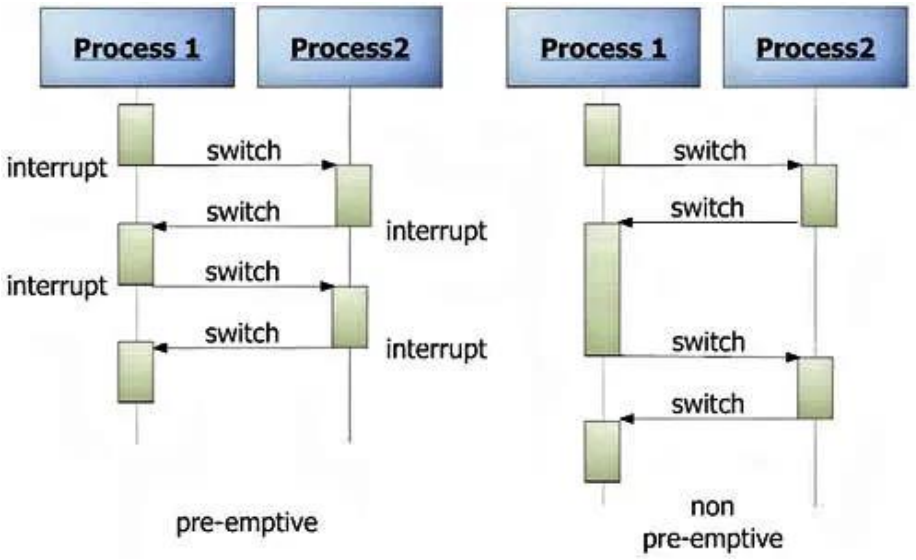

Рисунок 2.5 — Нерівномірне розподілення процесорного часу у моделі кооперативної багатозадачності

реалізувати певний спосіб продовжити виконання(continuation pattern). Існує декілька реалізацій цього шаблону. Це можуть бути, наприклад, корутини, функції зворотного виклику(callback function), кінцеві автомати та інші абстракції відкладених обчислень(Feature, IO, Task, TaskEither та інші).

Використання моделі кооперативної багатозадачності та планування задач на рівні процесу дозволяє використовувати механізми неблокуючого вводу-виводу, що значно ефективніші за звичайні блокуючі механізми.

2.6.2 Неблокуючий ввід-вивід

У програмах, які працюють у середі операційних систем, єдиний спосіб використовувати пристрої вводу-виводу є системні виклики. Системні виклики — це API, через яке забезпечується взаємодій програми з операційною системою та пристроями вводу-виводу. Для виклику системних викликів необхідно помістити параметри виклику у певні реєстри та викликати певний номер переривання, після чого керування перейде від програми до операційної системи, яка виконає певну роботу, помістить результат у певний реєстр або ділянку пам’яті та поверне керування до програми, що виконала виклик програми. Системні виклики використовуються для широкого спектр задач — від створення потоків до читання файлових дескрипторів. Оскільки головною платформою для розробляємого програмного забезпечення є Linux(операційна система сімейства UNIX), то саме на її прикладі буде розглянуто принцип неблокуючого виводу.

У операційних системах сімейства UNIX будь-який пристрій вводу-виводу є файлом і описується у операційній системі за допомогою файлового дескриптору, який містить всю інформацію про доступність, режим та стан пристрою або файлу. Якщо при запиті на зчитування даних пристрій ще не готов, у цьому режимі керування перейде до системи і буде заблоковано доки пристрій не буде готовим.

За замовчуванням всі файлові дескриптори створюються в синхронному режимі вводу/виводу, але якщо при визові системного виклику використати прапор O_NONBLOCK, то файловий дескриптор буде відкрито у неблокуючому режимі.

Після цього програма може зачекати певний час і після цього спробувати прочитати інформацію з пристрою знову[9]. На рисунку 2.6 а) зображено синхроний ввід-вивід, а на рисунку 2.6 б) зображено асинхронний ввід-вивід.

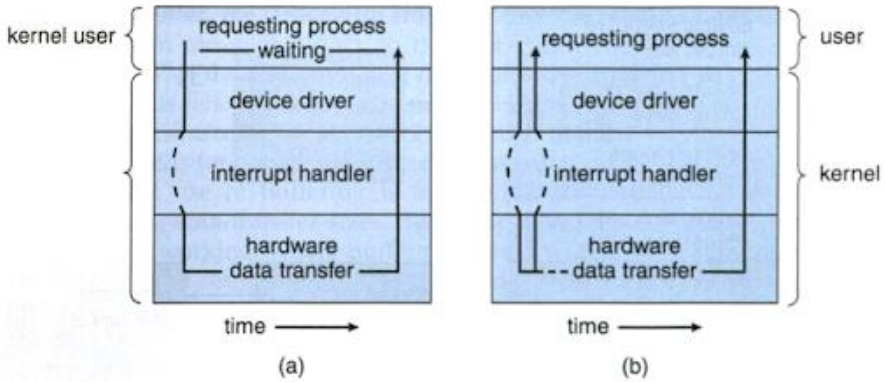

Рисунок 2.6 — Синхронний та асинхронний ввід-вивід

Асинхронний підхід добре працює, якщо кількість дескрипторів відносно мала. Якщо ж файлових дескрипторів десятки і більше, то використання неблокуючого вводу виводу хоч і надає переваги у порівнянні з блокуючою версією, але накладні витрати залишаються дуже високими. Для таких ситуацій існують особливі системні виклики — мультиплексори вводу-виводу, такі як select, epoll, kqueue. При використанні цих системних викликів ми можемо вказати набір файлових дескрипторів, з якими ми хотіли би працювати, наприклад зчитувати або записувати данні. Після цього процес заблокується, в загальному випадку, процес заблокується до готовності одного з файлових дескрипторів. Як тільки хоча б один файловий дескриптор буде доступний для обробки, процес продовжить свою роботу і зможе обробити доступні данні. У такому режимі працюють select та epoll в режимі level-triggered. Існують і інші режими, але ідея їх використання дуже схожа. Процес роботи з мультиплексуванням вводу-виводу зображено на рисунку 2.7

Використання мультиплексування дозволяє обробляти дуже велику кількість файлових дескрипторів з мінімальними накладними витратами процесорного часу.

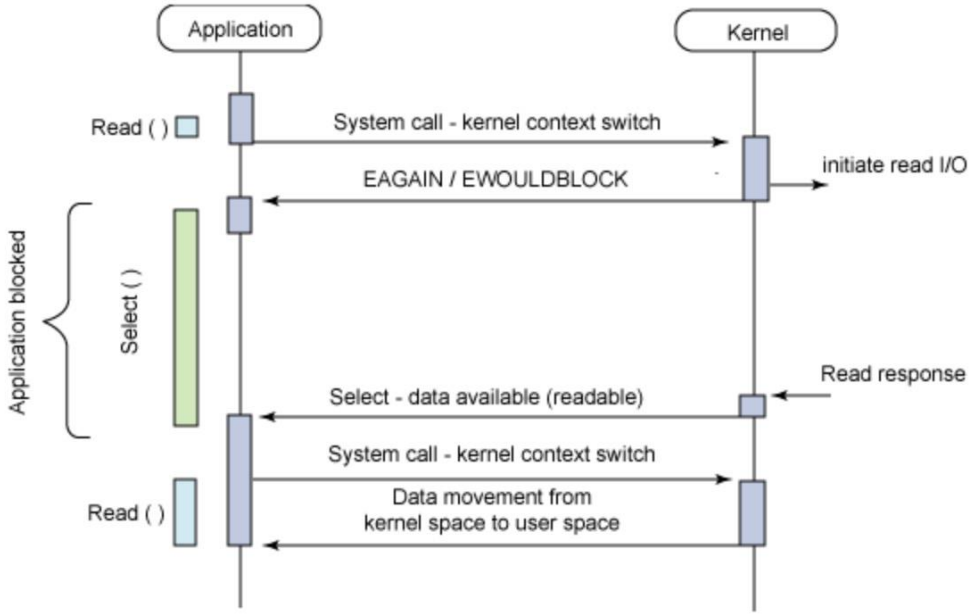

Рисунок 2.7 — Мультиплексування вводу-виводу

2.6.3 Приклади технологій, що базуються на неблокуючому вводі-виводі

Оскільки модель неблокуючого вводу-виводу дуже сильно відрізняється від традиційного блокуючого підходу, вона вимагає використання особливих парадигм. Кооперативна багатозадачність дуже добре працює у синергії з неблокуючим вводом-виводом.

Одним із перших комерційно успішних прикладів екосистеми, що цілком побудована навколо цих ідей стала Node.JS[10]. Це серверне середовище виконання JavaScript, яка базується на багатозадачності на основі циклу подій, також відомого як event loop (рисунок 2.8). Завдяки цьому використовується неблокуюча модель вводу-виводу з використанням функцій зворотнього виклику та промісів для реалізації паттерну продовження(continuation pattern).

Для синхронних операцій, що займають багато часу, такі як криптографія, Node.JS використовує пул потоків.

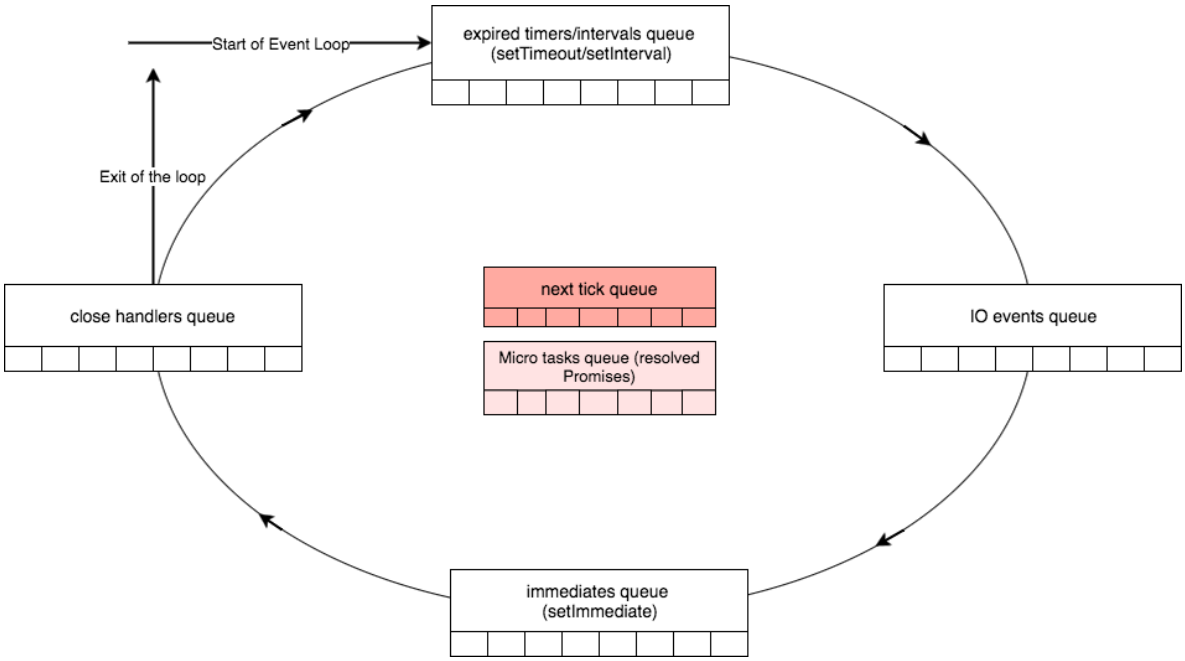

Рисунок 2.8 — Схема циклу подій у Node.JS

З іншого боку, мова програмування Go обрала іншу модель[11]. Всі функції в go — корутини(в самому Go їх називають горутини) і можуть призупиняти виконання. Горутини реалізовують концепцію “зелених” потоків і додають мінімальні накладні витрати. Go використовує work-stealing планувальник. Він може працювати з невеликою кількістю реальних потоків і балансувати навантаження між декількома обчислювальними ядрами. Модель планувальника зображено на рисунку 2.9

Варто зазначити, що дуже часто Go запускають на одному фізичному потоці, щоб запобігти блокувань, оскільки це значно підвищує швидкодію і знижує накладні витрати. У такому режимі робота Go дуже схожа на цикл подій Node.JS.

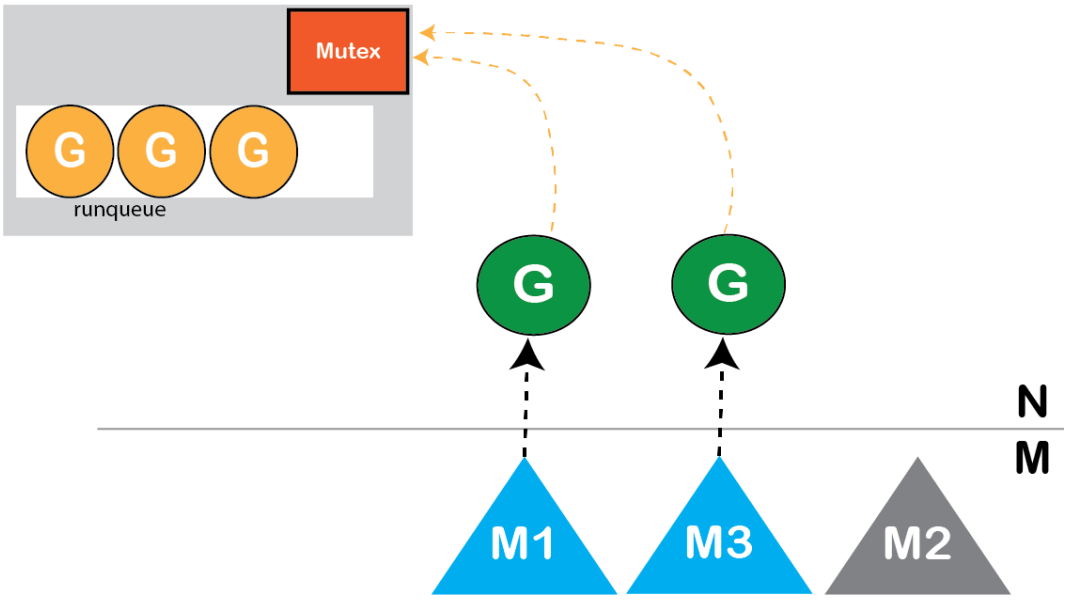

Рисунок 2.9 — Work-stealing планувальник Go

2.6.4 Бібліотека Akka

Akka — дуже популярна Scala та Java бібліотека для реалізації обчислень з використанню моделі акторів[12]. Модель акторів — це математична модель, яка лежить в основі моделі ООП — взаємодії незалежних акторів через обмін повідомлень без використання глобального стану. Ця модель надає змогу проводити потокобезпечні обчислення і запобігає велику кількість помилок зв'язаних з спільними між потоками даними, таких як data race та race condition. Також Akka бере на себе низькорівневі операції вводу-виводу, надаючи асинхронне API для роботи з файлами, сокетами та мережевими запитами. Використання цієї бібліотеки значно знизить трудомісткість при побудові брокера і дозволить сконцентруватись на імплементації протоколу, а не роботі з вводом-виводом.

2.6 Пакетний менеджер

Пакетний менеджер — програмне забезпечення, яке дозволяє керувати залежностями та компіляцією програми. Використання пакетних менеджерів значно полегшує розробку і супроводження програмного забезпечення.

Використання пакетного менеджера надає відносно невеликі переваги у малих командах, але при рості розміру команд або їх географічному розподіленні, переваги пакетного менеджеру стають дуже значними. Так як подальший розвиток програмного продукту планується за моделлю програмного забезпечення з відкритим вихідним кодом, то використання пакетного менеджеру є обов'язковим.

На сьогоднішній день в екосистемі Java активно використовуються два пакетні менеджери: Maven та Gradle. Ці пакетні менеджери активно використовуються у багатьох проєктах.

Maven — відносно старий пакетний менеджер. Використовує XML для конфігурацій, досить легковісний, але не підтримує ітеративну компіляцію. Перевірений часом, але й морально застарілий — це добрий вибір якщо Maven вже активно використовується в середині компанії або при підтримці легасі проєктів.

Gradle — доволі молодий пакетний менеджер. На відміну від Maven використовує DSL на основі Kotlin або Groovy. Gradle — це більше ніж просто пакетний менеджер, це також і інструмент для будування проєкту[13]. Гнучка система задач дозволить сконфігурувати майже будь-який процес будівництва та легко підключити додаткові можливості, такі як контейнеризація, бандлінг та інше. Можливості багатопроєктного будування та конфігурації будуть дуже актуальними у проєктах з декількома компонентами. Gradle — хороший вибір як для нових проєктів, так і у випадку якщо проєктів на основі Maven не багато — повільна міграція на Gradle дасть доступ до широких можливостей Gradle та поліпшить час будування, що позитивно вплине на продуктивність розробників та час розгортання проєкту.

Проаналізувавши доступні варіанти використання Gradle у якості пакетного менеджера було обрано більш перспективним. На це вплинуло дуже багато факторів, головні з них:

— моральна застарілість Maven;

— відсутність застарілих компонентів;

— відсутність інших Maven проєктів;

— гнучка система будування;

— мульти проєктне будування;

— незручність XML як формату конфігурації;

— ітеративна компіляція;

— активний розвиток та підтримка суспільством.

Висновки до розділу

Було проаналізовано основні вимоги до програмного забезпечення та знайдено технічні рішення, які значно полегшать розробку як брокеру, так і клієнту.

Для того, щоб отримати кросплатформність та високу швидкодію не жертвуючи безпекою ПЗ з невеликими накладними витратами було обрано мову програмування Java, яка виконується на віртуальній машині JVM

Для побудови веб-інтерфейсу буде використовуватись сучасний MVC фреймворк Spring, що значно полегшить розробку панелі керування брокером.

Для обробки високої кількості одночасних повідомлень буде використовуватись асинхронне API разом з неблокуючим вводом-виводом. Для запобігання помилок при роботі з системними викликами буде використано бібліотеку Akka, яка впроваджує модель акторів для конкурентних розподілених обчислень.

Для полегшення подальшої розробки та підтримки з використанням моделі програмного забезпечення з відкритим вихідним кодом буде використано пакетний менеджер Gradle.

3 СПЕЦИФІКАЦІЯ ПРОТОКОЛУ

3.1 Життєвий цикл протоколу

На відміну від TCP, який є сесійним протоколом, UDP не використовує механізм сесій, тому сесії будуть реалізовані на транспортному рівні додатку. Кожна сесія має ідентифікатор та прив'язана до певної IP адреси. У поточній версії протоколу можливості широканальної передачі не використовуються, але планується створення ширококанальної передачі для режиму з негарантованою доставкою у наступних версіях. Для ініціалізації сесії використовується TCP-подібний трьох ступеневий хендшейк.

Спочатку клієнт відправляє запит на створення сесії. При створенні сесії необхідно вказати тип під'єднання: консьюмер або продюсер. В обох випадках необхідно вказати чергу, з якою клієнт збирається працювати, а у випадку консьюмера — вказати, які партиції черги він буде прослуховувати. При створені сесії передається інформація про авторизацію та шифрування.

У поточній версії протоколу підтримується лише симетричне шифрування, але у наступних версіях протоколу планується додавання асиметричного шифрування, підпису даних для перевірки цілісності та вимикання шифрування для мало важливих даних, таких як інформація з сенсорів.

Для авторизації використовується комбінація логіну та паролю. Користувач створюється у панелі керування і може мати різні права, наприклад можна обмежити можливість прослуховувати та надсилати повідомлення. Можна дозволити комунікацію з чергами, дозволивши працювати з усіма чергами, або лише з певним переліком сесій. Для створення користувача необхідно мати привілеї адміністратора. При створені екземпляру брокеру за замовченням створюється користувач з привілеями адміністратора, використавши який можна адмініструвати подальшу работу брокеру, наприклад створювати користувачів та черги.

Дані аутентифікації та інформація про чергу партицій та режим роботи передаються у тілі запиту на створення сесії. Для їх серіалізації використовується формат JSON. Використання цього формату не є оптимальним, але оскільки ця операція відбувається не дуже часто, то десеріалізація не вплине на швидкість роботи протоколу.

Для підтвердження автентифікації брокер відправляє пакет підтвердження, який може мати декілька статусів:

— 1 — автентифікацію встановлено успішно. З цього моменту клієнт може відправляти повідомлення, якщо він був створений у режимі продюсера або отримувати їх, якщо він був створений у режимі консьюмеру;

— 2 — помилка автентифікації. При створені не вдалось знайти користувача або пароль не збігається, у автентифікації відмовлено;

— 3 — помилка авторизації. У користувача не достатньо прав для роботи з чергою у зазначених режимах;

— 4 — збій брокеру. У цьому випадку на брокері відбулась виключна ситуація, що унеможливила подальшу підтримку під'єднання.

У випадку помилкового статусу відповідь також містить JSON з додатковою інофрмацією про помилку. Це дозволить детальніше зрозуміти що призвело до помилки та спробувати переприєднатись якщо причиною стали проблеми мережі.

У випадку успіху, відповідь містить JSON з інформацію про налаштування брокеру, таку як — тайм-аут під'єднання.

Якщо клієнт не отримав підтвердження створення сесії він спробує переприєднатись знову. Клієнт зробить декілька спроб, після чого повідомить про неможливість під'єднання до серверу.

Після успішної спроби клієнт відправляє декілька EXO для встановлення значення RTT.

Після того як з'єднання встановлено, необхідна його постійна підтримка. Для цього клієнт та сервер використовують обмін EXO пакетами. EXO пакети мають подвійне призначення: вони дозволяють встановити коректне значення

RTT та перевірити доступність елементів системи. При створені екземпляру серверу для нього вказуються такі значення, як таймаут під'єднання та кількість спроб при відправлені повідомлення. Якщо клієнт не відправить EXO пакет протягом вказаного таймауту, то сервер вважає що клієнт не відповідає і від'єднує його, термінуючи сесію. При обміну EXO пакетами клієнт та сервер оновлюють значення RTT, що дозволяє більш точно контролювати час очікування.

Протягом часу існування сесії клієнт та сервер обмінюються повідомленнями. Обмін відбувається у напів дуплексному режимі: хоч і сервер і клієнт відсилають один одному повідомлення на підтвердження отримки, на логічному рівні відбувається обмін лише у одну сторону: від клієнта до брокеру у випадку продюсеру і від брокеру до клієнта у випадку консьюмера.

При від'єднанні клієнт або сервер просто перестає відправляти EXO пакети. Після цього сервер термінує сесію через неактивність.

Життєвий цикл протоколу зображено на кресленику ІТ61.110БАК.004Д1, а структуру пакетів протоколу — на кресленику ІТ61.110БАК.004Д2

Загалом ця структура протоколу досить проста, але дозволяє ефективно з низькими накладними витратами обробляти велику кількість під'єднання. У наступних розділах буде детальніше розглянуто різноманітні аспекти протоколу

3.2 RTT

RTT — round trip time, час за який повідомлення доходить від клієнта до серверу і назад. Цей час дозволяє зрозуміти чи достатньо часу пройшло з часу відправки повідомлення і чи необхідно відправляти його знову. Постійне відправлення EXO пакетів дозволяє корегувати RTT і отримувати більш точні результати відправки, тим самим знижуючи навантаження на мережу. EXO пакети відсилають на сервер поточний RTT обрахований на клієнті, щоб гарантувати актуальне значення і підвищити точність роботи протоколу.

Подібні механізми використовуються у TCP, але там вони мають більш складні алгоритми з високими накладними витрати. Використання простіших алгоритмів дозволить трохи підвищити пропускну здатність на знизити затримки.

3.3 Гарантія доставки

Гарантія доставки — важлива властивість будь-якого транспортного протоколу. Хоч UDP і не гарантує доставку, ми можемо створити механізми контролю цілісності на рівні самого додатку. Протокол обміну повідомленнями складається з двох частин: транспортної і логічної. Транспортна частина контролює такі значення як номер повідомлення і відповідає за підтвердження доставки. Логічна ж частина відповідає за роботу додатку — брокеру та клієнту.

При створені транспортної частини було проаналізовано декілька вимог і можливих застосувань. Основним з них є використання у середі сучасних датацентрів або інтернету речей. У прототипі протоколу основну увагу буде звернуто на роботу у датацентрах. Принципіальною різницею між цими двома застосуваннями є процент втрачених пакетів та пропускна здатність.

Сучасні датацентри обладнані високошвидкісними мережами з високою пропускною здатністю та надійністю, що дозволяє отримати дуже низький відсоток втрачених пакетів, близький до десятих відсотка[1], тому втрата пакетів буде дуже рідким явищем. До того ж пропускна здатність датацентрів дозволяє передавати повідомлення цілком, а не розбивати його на декілька частин. Це дозволить запобігти потреби відновлення черги пакети через втрату порядку, що значно підвищить пропускну здатність протоколу.

Інша можлива область використання протоколу є інтернет речей. У цьому випадку не завжди буде існувати стабільне з'єднання з мережею, активно використовуються як дротові так і бездротові канали передачі даних для подальшої обробки.

Сучасні дротові канали передачі досить стабільні і надійні, тому використовувати їх можна у тому ж режимі, що й дата центр. Ситуація з бездротовими каналами зв'язку трохи інша — сучасні технології дозволяють значно поліпшити ситуацію з втратою пакетів і досягнути близько двох відсотків втрачених пакетів[2].

З іншого боку особливість рішень сфери інтернету речей полягає у тому, що втрата одного або двох пакетів не є критичною — дуже часто сенсори надсилають десятки повідомлень на секунду, і у разі якщо хоча б один з них дійде до цілі — сенсор виконає свою роботу коректно і важливу інформацію не буде втрачено. Тому у клієнтській частині досить актуальною буде можливість відправити декілька агрегованих пакетів і отримати підтвердження доставки на хоча-б один із них. Це дозволить значно підвищити ефективність рішення при роботі з сенсорами та датчиками, які під'єднані до мережі з використанням бездротових мереж.

У загальному випадку транспортний рівень протоколу буде працювати аналогічно TCP — на кожне отримане повідомлення необхідно буде надсилати підтвердження. Аналогічний підхід використовується у протоколі QUIC, який розробляється компанією Google та реалізує HTTP протокол використовуючи UDP у якості транспортного рівня. Підтвердження містить номер повідомлення, який було отримано. В загальному випадку схема роботи зображена на рисунку 3.1. При успішній відправці отримувач повідомляє користувача про успіх. Це дуже простий випадок, який буде траплятись більшу частину відправок. Надійність сучасних мереж і цілісність повідомлень дозволить значно зекономити на перевірці цілісності. Але оскільки протокол черг повинен гарантувати послідовність доставки повідомлень, то все одно доведеться відновлювати послідовність логічних повідомлень(а не фрагменти одного повідомлення, як це відбувається у TCP).

Менш поширений випадок зображено на рисунку 3.2.

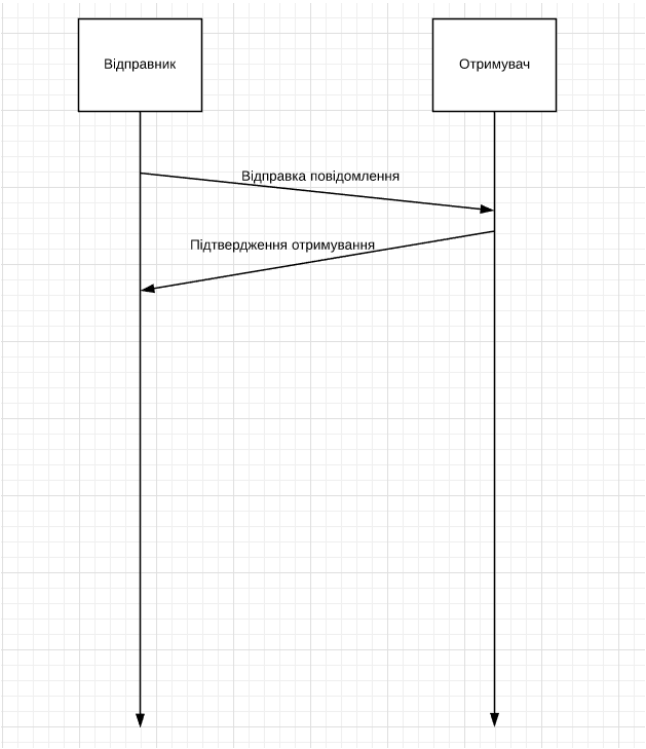

Рисунок 3.1 — Успішна доставка повідомлення

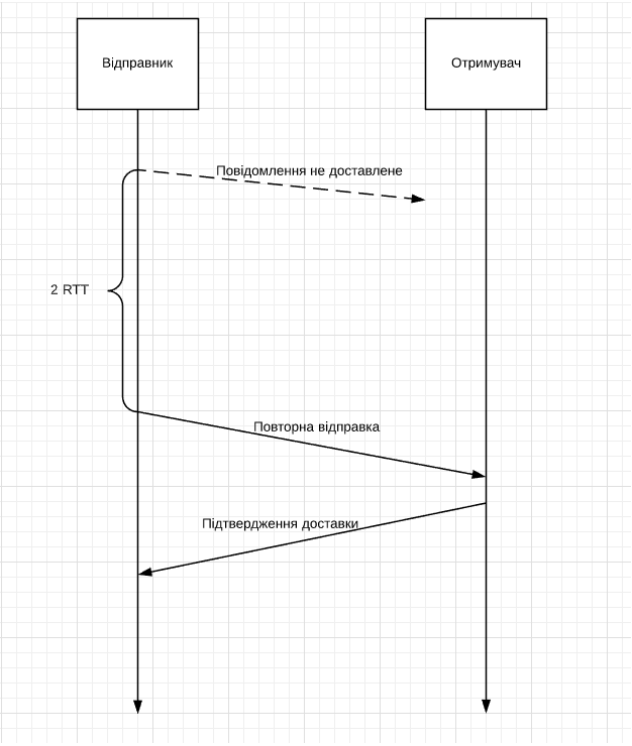

Рисунок 3.2 – Повторна відправка повідомлення

У цьому випадку повідомлення не надходить до отримувача. При цьому буде відіслано ще один пакет, який дійде і отримає підтвердження. Повторне

повідомлення буде відправлено через 2 RTT. Наступні повідомлення будуть надсилатись з затримкою, що наростає по експоненті. Наступне повідомлення буде відправлено через 4 RTT, потім 8RTT і після цього 16RTT. Тому не варто встановлювати кількість спроб відправки більше ніж 5.

У цьому випадку повідомлення не надходить до отримувача. При цьому буде відіслано ще один пакет, який дійде і отримає підтвердження. Повторне повідомлення буде відправлено через 2 RTT. Наступні повідомлення будуть надсилатись з затримкою, що наростає по експоненті. Наступне повідомлення буде відправлено через 4 RTT, потім 8RTT і після цього 16RTT. Тому не варто встановлювати кількість спроб відправки більше ніж 5.

Далі розглянемо ситуацію, коли два пакети відправляються до брокеру, і один з них надходить, а інший - ні. При цьому пакету будуть вимагати перестановки порядку. Таку ситуацію розглянуто на рисунку 3.3.

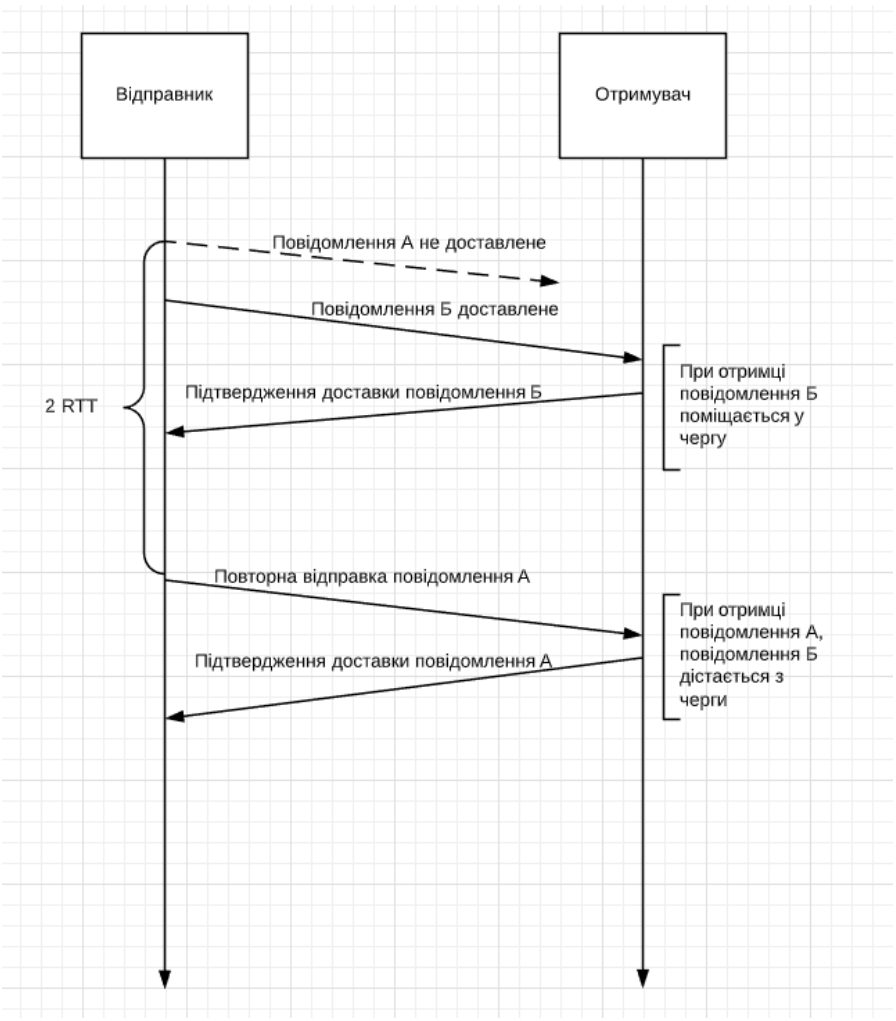

Рисунок 3.3 — Відновлення порядку при отриманні повідомлень

На рисунку 3.4 розглянуто надсилання двох повідомлень А і Б. Нехай порядковий номер повідомлення А буде 100, а номер Б — 101. Отримуюча сторона слідкує за номером останнього отриманого повідомлення, і якщо отримане повідомлення не задовільняє порядку — воно складається в мінімальну двійкову купу ─ сортовану структуру даних, яка є найбільш ефективною у цьому випадку. Після отримання повідомлення з кучі дістаються всі повідомлення, що задовільняють порядку.

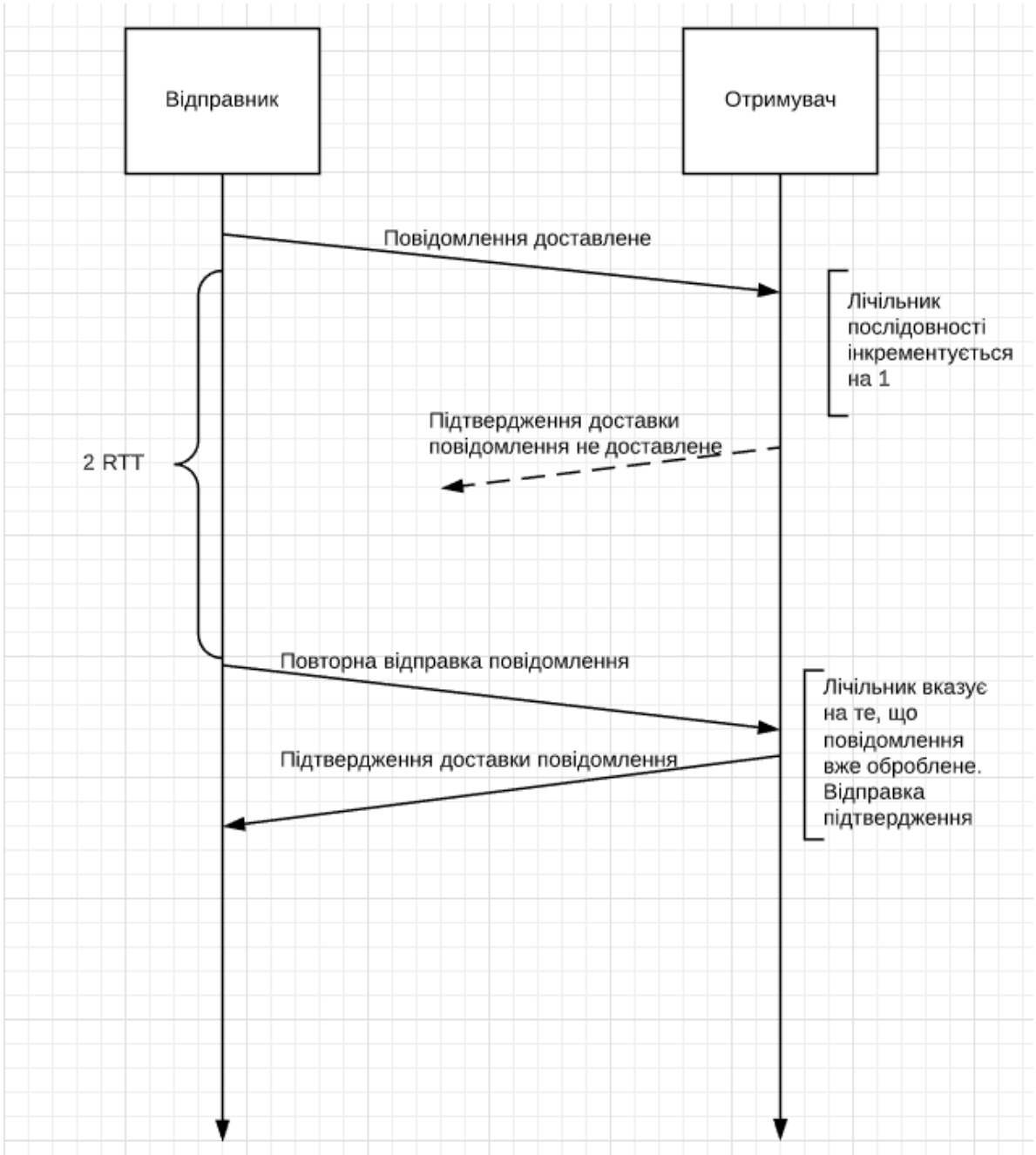

Рисунок 3.4 — Втрата повідомлення про підтвердження

Використання купи у цьому випадку є дуже вигідним рішенням, оскільки повідомлення будуть надходити у приблизно відсортованому порядку і отримувати данні з кучі теж будуть у певному порядку — від найменшого порядкового номер до найбільшого.

Оскільки купа — відсортована структура даних вона забезпечує швидкий пошук — асимптотична складність пошуку у кучі дорівнює O(log n), що є задовільним результатом. Також можлива ситуація, коли повідомлення було доставлене до отримувача, а підтвердження про доставку — ні. Цю ситуацію розглянуто на рисунку 3.5.

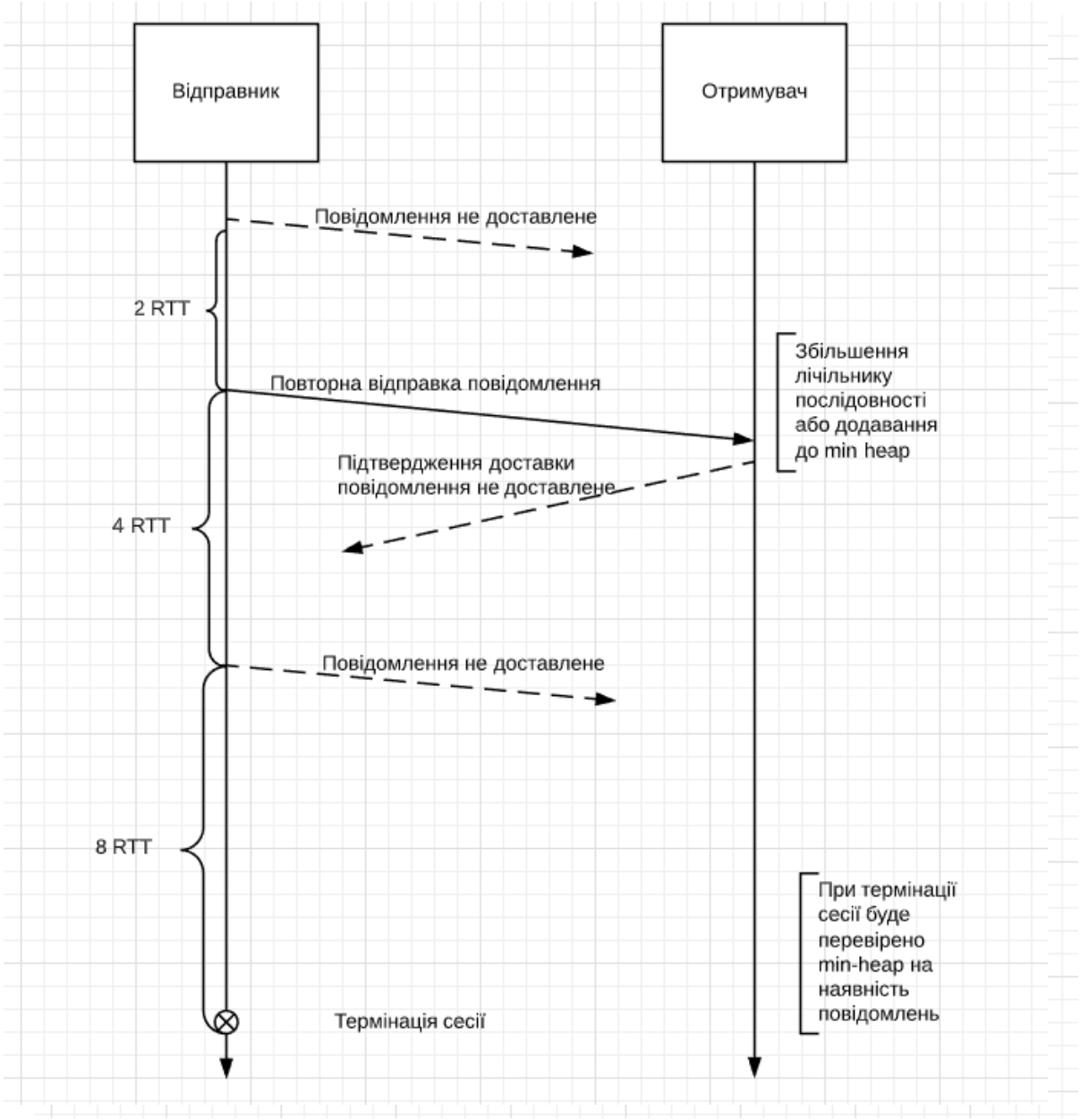

Рисунок 3.5 — Термінація сесії через не підтверджену доставку

У разі, якщо повідомлення було доставлене, то через 2 RTT відправник знову спробує відправити повідомлення до отримувача. Далі можливі два варіанти: отримане повідомлення змінило лічильник, оскільки було отримано у правильному порядку або отримане повідомлення було додано у чергу очікування. У першому випадку при отримані повідомлення лічильник вкаже на те, що повідомлення вже прийнято в обробку і одразу буде надісланий пакет підтвердження. У разі, якщо пакет порушив послідовність, буде спроба додати пакет до купи, але оскільки він там вже існує, то його додавання проігнорують і буде відправлено підтвердження. Такий алгоритм дозволяє обробляти повідомлення з мінімальними накладними витратами.

При розробці протоколу важливо буде приділити увагу паралельній обробці та стану гонки. Для того, щоб транспортний рівень був потокобезпечним необхідно забезпечити коректне блокування, оскільки використання алгоритмів без блокування(lock-free) у цьому випадку не є можливим. Тому для блокування варто використати RWLock(мьютекс блокування на читання та запис), що дозволить отримати більш гранулярну роботу з ресурсами. З іншого боку такий підхід додасть додаткові зайві витрати процесорного часу, і якщо забезпечити потокобезпечність на рівні сесії можна значно підвищити швидкодію алгоритму, особливо при високих навантаженнях.

Остання важлива ситуація — коли мережа не стабільна і пакет відправити не вдалося. У такому випадку пакети просто не надходять до отримувача. При цьому після скінчення ліміту спроб повторної відправки повідомлень, клієнт розірве сесію і перестане надсилати EXO пакети. Подібною до цієї є ситуація, коли при одні зі спроб пакет було доставлено, але підтвердження так і не дійшло. При цьому можливі два випадки: отримане повідомлення відповідає порядку або не відповідає порядку. У будь якому разі при знищені сесії всі дані з мінімальної кучі передаються далі у відсортованому вигляді, що дозволяє додати їх до черг і таким чином запобігти втрату даних.

3.4 Контроль заторів у мережі

Алгоритми контролю заторів дозволяють зменшити навантаження на мережу і запобігти циклу «смерті», який остаточно унеможливлює використання мережі.

Цикл «смерті» це явище, коли мережа перевантажується спробами повторної відправки даних при відсутності підтвердження. Так, наприклад, якщо причиною збою став пристрій з низькою пропускною здатністю, спроба перевідправити пакети ще більше перевантажить пристрій і рано чи пізно призведе до його перевантаження, що переведе трафік до інших вузлів, які теж можуть перевантажитись і викликати каскад відмов у всій мережі. Для запобігання цього у протоколах використовують різні механізму контролю заторів.

Оскільки основна область використання протоколу у поточній реалізації — датацентри з надійним обладнанням, то достатньо використовувати досить простий механізм контролю заторів — перевідправляти спроби за експонентою степені двійок. Перша повторна спроба буде надіслана через 2 RTT, друга — через 4 RTT, третя — через 8 RTT і так далі. Такий підхід не дуже добре працює при передачі інформації на великі відстані з ненадійними каналами зв'язку, оскільки значно підвищує затримку при втраті пакетів, але у датацентрах з малим RTT це дуже не критично впливає на затримку.

3.5 Серіалізація

Серіалізація змісту повідомлень — важлива частина будь-якої системи. Серіалізація використовується для представлення даних у форматі, придатному для передачу через мережу або інші засоби комунікації. Часто, дані серіалізуються у структурований масив байт або у строку. Для своїх інфраструктурних цілей протокол використовує або фіксовану структуру з офсетами або використовує формат JSON. Для тіла повідомлення за

замовченням також використовується JSON серіалізація, але її можна замінити на іншу, більш задовільну для потреб додатку, наприклад Apache Thrift або Protobuff.

3.6 Шифрування

Для шифрування буде використовуватись симетричне шифрування. Ключ створюється при створені черги. Ключі розповсюджуються за допомогою протоколу Діффі-Хелмена.

Використання симетричного шифрування дозволяє впровадити так зване end to end(скрізне) шифрування. Це дозволить підвищити безпечність даних у транзиті і знизити накладні витрати на шифрування/дешифрування.

Використання end to end шифрування має декілька переваг і недоліків. До переваг відносять:

— відсутність накладних витрат на дешифрування на проміжному вузлі;

— захищеність даних при передачі.

Але використання скрізного шифрування має певні недоліки:

— вразливість до атак типу «людина у середині»(man in the middle);

— атака кінцевих точок;

— вразливість ключів до брут-форсу.

Такі недоліки не є критичними у високо захищених датацентрах, але при роботі з інтернетом речей інформаційна безпека — дуже актуальна проблема, тому у наступній версії протоколу планується використання скрізного шифрування, але замість симетричного буде використано асиметричне шифрування. Також планується можливість вимикання шифрування для підвищення швидкості обробки у кінцевих вузлах та можливість використання цифрового підпису змісту повідомлення для запобігання модифікації.

Також існує проблема брут-форсу ключів, оскільки ключ створюється для сесії, то у наступних версіях необхідно продумати можливість автоматичного

оновлення ключів для сесії з певною періодичністю, що допоможе запобігти перебору можливих ключів.

Серед симетричних алгоритмів шифрування найбільш перспективним для використання є алгоритми DES та AES. Це блокові шифри, тому якщо повідомлення не кратне довжині блоку, воно доповнюється порожніми символами. Це може стати проблемою при використані у системах з дуже малим розміром повідомлень(менше 100 біт), оскільки тоді багато місця повідомлення буде використано нераціонально. Для першої версії протоколу буде використано більш сучасний алгоритм AES з довжиною блоку 128 біт і довжиною ключа 256 біт.

Висновки до розділу

Резюмуючи, можна сказати, що транспортний рівень протоколу ще не готовий до використання у корпоративних додатках. Хоч протокол і реалізує більшість необхідних алгоритмів вони не є достатньо опрацьованими і необхідний їх подальший розвиток. Навантажувальне тестування дозволить визначити вузькі місця і чіткіше зрозуміти які частини протоколу необхідно допрацювати.

Протокол впроваджує механізм гарантованої доставки, який також гарантує послідовність доставки. У якості буферу на стороні отримувача використовується мінімальна двійкова куча, яка є оптимальною структурою для збереження відсортованих даних.

Протокол впроваджує базовий механізм контролю заторів. Він працює на основі експоненційної затримки при повторній відправці недоставлених повідомлень.

На логічному рівні протокол впроваджує можливість серіалізації тіла повідомлення з використанням різних протоколів. За замовчуванням використовується JSON, але можливе використання таких протоколів серіалізації як Apache Thrift та Protocol Buffers.

Для шифрування використовується симетричні алгоритми з end to end(скрізним) шифруванням. Для розподілення ключів використовується протокол Діффі-Хелмана. Для шифрування буде використовуватись сучасний блоковий шифр AES з довжиною ключа 256 біт.

4 АРХІТЕКТУРА ПРОГРАМНИХ КОМПОНЕНТІВ

4.1 Архітектура брокера

4.1.1 Загальний опис компонентів брокеру

Брокер — серверна частина рішення для обміну повідомленнями. Він виступає у якості посередника між продюсером та консьюмером та виконує такі задачі як мультиплексування, демультиплексування, маршрутизація та збереження повідомлень. Також він слідкує за логічним виконанням правил протоколу: не більше одного слухача для кожної партиції, продюсер може надсилати повідомлення лише до однієї черги та інші обмеження.

Брокер використовує 3 API для взаємодії з зовнішнім світом: HTTP протокол при використанні панелі керування, UDP протокол для взаємодії з продюсерами та консьюмерами та API файлової системи для серіалізації змісту партиції до персистентного сховища даних, таких як жорсткий диск.

Брокер складається з 5 основних компонентів: панелі керування, менеджеру черг, менеджеру сесій, серіалізатору та мультиплексору вводу-виводу. Ці компоненти не залежать один від одного і взаємодіють через такі API як QOA(queue operation API), QTP(queue transport protocol) та протокол серіалізації. Використання цих API дозволяю значно підвищити самостійність компонентів системи і знизити ступінь зв’язності(coupling) між компонентами системи.

Загальну архітектуру брокера зображено на кресленику ІТ61.110БАК.004Д3

4.1.2 Панель керування

Панель керування відповідає за адміністрування екземпляром брокеру. Вона робить можливим такі операції як створення черг та користувачів. Цей компонент впроваджує інтуїтивно зрозумілий користувацький веб-інтерфейс,

що значно знижує час, який необхідно витратити для отримання навичок адміністрування брокером.

Вона реалізована з використанням сучасного веб фреймворку Spring з використанням архітектурного шаблону MVC. Для реалізації браузерного інтерфейсу використовуються стандарти HTML5 та CSS3. Для програмування логічної частини браузерного додатку використовується мова програмування JavaScript.

Додаток використовує класичну схему відображення на стороні серверу, оскільки більш сучасні підходи, такі як односторінкові додатки(single page application) та гібридні рішення додадуть зайву складність до додатку.

Панель керування дозволяє створювати нових користувачів і черги. У поточній версії брокеру не існує можливості додати обмеження користувачам на прослуховування або відправку повідомлень лише до певних черг.

Панель керування взаємодіє з зовнішнім світом за допомогою протоколу HTTP/1.1. У наступних версіях планується перехід на HTTP/2. Також існує можливість використовувати шифрування HTTPS. Для цього необхідно додати ключі та змінити налаштування у коді брокеру.

За замовчуванням панель керування використовує порт 8080, але це можна змінити у налаштуваннях. Перебудовувати проєкт брокеру при цьому не треба.

Для створення та взаємодії з чергами використовується інтерфейс Queue Operation API, який буде детальніше розглянуто у розділі 3.2.3

Загалом можна сказати, що компонент панелі керування вийшов досить незалежним і не зв’язаним з іншими компонентами.

4.1.3 Менеджер черг

Менеджер черг є серцевим компонентом брокеру. Він відповідає за роботу черг та партицій. Саме він слідкує за валідністю інваріантів черг, а саме за станом відправки та отримання повідомлень.

Загальну схему архітектури менеджеру черг зображено на рисунку 4.1. На ньому зображені компоненти менеджеру сесій, а саме: адаптер Queue Operation API, адаптер Queue Transport Protocol, агрегатор черг, черги та партиції.

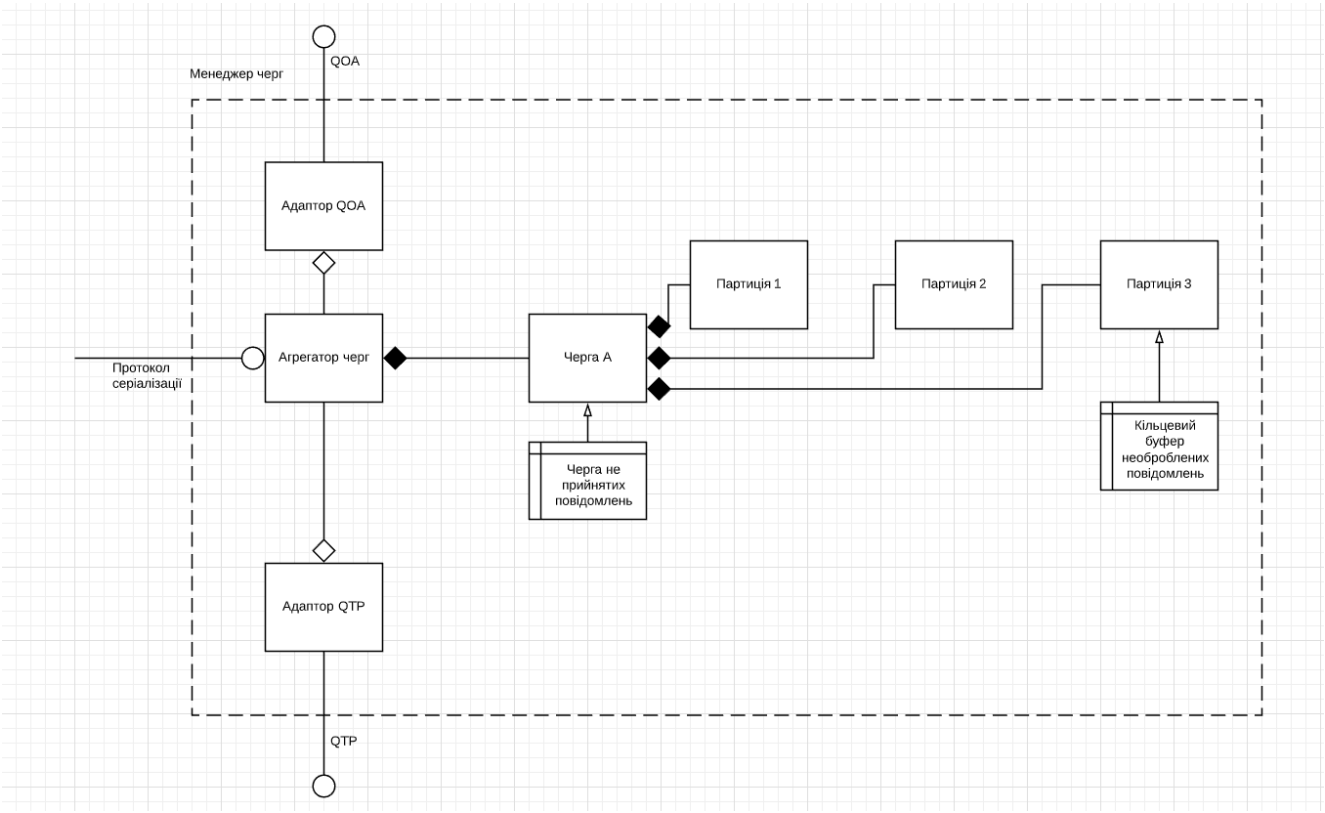

Рисунок 4.1 — Архітектура менеджеру черг

Агрегатор черг — компонент, що відповідає за зберігання та серіалізацію сесії. Він виступає у якості проксі при роботі з чергами та виконує задачу маршрутизації при запитах до конкретної сесії. Сам по собі агрегатор черг не містить стану, окрім списку черг.

Агрегатор черг використовує протокол серіалізації для зберігання черг у персистентному сховищі даних. Агрегатор черг вступає у відносини агрегації з усіма чергами(при видаленні агрегатору знищуються всі черги). Агрегатор черг взаємодіє з адаптерами QOA та QTP через відношення композиції.

Адаптери QOA та QTP виконують функцію делегації для реалізації відповідних інтерфейсів. Вони дозволяють зробити доступ до агрегатору більш гранулярним та контрольованим. Також вони дозволяють задовільнити

принципу сегрегації інтерфейсів, що полегшить модифікацію кода у майбутньому.

Черги та партиції — основна абстракція протоколів асинхронного обміну повідомленнями. Вони представляють односторонній потік повідомлень, які генерують продюсери та отримують консьюмери. У поточній реалізації брокеру черга виступає як незалежний актор і сама слідкує за своїм станом. Основними функціями черги є:

отримувати нові повідомлення та розподіляти їх по партиціям;

надіслати повідомлення з партиції на передачу транспортному рівню;

слідкувати за інваріантами прослуховування партицій;

сповіщати партицію про доставку і обробку повідомлень та готовність користувачів до отримання наступного повідомлення.

Для отримання повідомлень черги використовують FIFO структуру даних чергу(queue). Для гарантії послідовності використовується механізм блокування. Після прочитання повідомлення з черги, воно делегується до партицій.

Кожна партиція використовує циклічний буфер для зберігання повідомлень, таким чином можливе перемотування часу, пересуваючи вказівник на поточне повідомлення у буфері. Кількість збережених повідомлень можна налаштувати у ручну, таким чином регулюючи на скільки можна повернутись назад.

Циклічний буфер — структура даних, що дуже активно використовується для організації FIFO обміну повідомлень. Цю структуру даних активно використовують як у додатках корпоративного рівня, так і при програмуванні мікроконтролерів.

Основною його перевагою є простота реалізації, зручність у використанні та висока продуктивність.

4.1.4 Серіалізатор

Компонент серіалізатору відповідає за персистентне збереження черг і партицій. Серіалізатор впроваджує API протоколу серіалізації, який у свою чергу використовується у менеджері черг.

Для серіалізації буде використано механізм синхронізації змісту диску зі змістом черги. Для цього буде використовуватись алгоритм знаходження різниці між змістом партицій та змістом диску. Кожна партиція буде серіалізуватись у окремий файл. Подібний механізм використовують деякі сучасні бази даних, наприклад MongoDB та Redis.

У поточній версії брокеру серіалізацію не реалізовано. Основною проблемою стала складність реалізації алгоритму різниці та роботи з файловою системою. Також при реалізації було виявлено проблеми механізму синхронізації: дані можуть бути втрачені при збою програми до запису на жорсткий диск. Це стане більшою проблемою при роботі з розподіленою системою, тому необхідно детальніше розробити механізм серіалізації у наступній ітерації розробки та проаналізувати його з урахуванням CAP теореми.

4.1.5 Менеджер сесій та мультиплексор вводу-виводу

Менеджер сесій та мультиплексор вводу-виводу відповідають за логічну і фізичну роботу з вводом-виводом. Менеджер сесій представляє логічну частину роботи з сесіями і є посередником між чергою та вводом-виводом. Мультиплексор вводу-виводу є комбінацією UDP серверу та клієнту. Клієнт відповідає за транспортний рівень протоколу та доставку повідомлень. Його буде детальніше розглянуто у розділі 3.3. Сервер відповідає за отримання та парсинг повідомлень протоколу. Цикл вводу-виводу працює у одно поточному режимі, а парсинг та подальше обробка делегується акторам. Парсинг

повідомлень відбувається на рівні мультиплексору, після чого повідомлення передаються на подальшу обробку на рівень сесій.

Сесії представляють логічну одиницю взаємодії з продюсерами та консьюмерами. Сесії — самостійні актори і містять в собі інформацію про користувача, режим та інше.

4.1.6 Виявлені проблеми брокеру

Брокер має велику кількість недоліків, які необхідно врахувати у наступних версіях.

Проблема з персистентністю залишається дуже гострою: по-перше алгоритми знаходження різниці є досить складним у реалізації і додасть додаткові накладні витрати. Також підтвердження отримки повідомлення вправляється у не залежності від успішності додавання повідомлення до черги або зберігання їх на диску.

Також з’явилась проблема відсутності зворотнього зв’язку між продюсером та консьюмером. У випадку, якщо консьюмер генерує дані швидше, ніж консьюмер здатний їх обробляти, надлишок повідомлень скупчується на сервері і призводить до блокувань кільцевих буферів. Також відсутній механізм backpressure, що дозволяв би зменшити навантаження на брокер та консьюмерів.

Зберігання всіх повідомлень в оперативній пам’яті дозволяє значно підвищити швидкість роботи, але ціною значних витрат пам’яті. У випадку, якщо повідомлення генеруються дуже швидко і хочеться зберігати повідомлення за декілька днів, їх просто можна не вмістити в оперативну пам’ять. У такому випадку слід урахувати можливість зберігання проекції частини буферу у оперативній пам’яті, а іншу частину зберігати на жорсткому диску. Такий підхід дозволить значно зменшити витрати оперативної пам’яті.

Також у майбутніх версіях слід подумати над можливістю використання неблокуючих структур даних, що може значно знизити час блокувань і тим

самим збільшити пропускну здатність рішення. Також можливо слід подумати про реорганізацію брокеру і використання іммутабельних структур даних, що може полегшити подальшу розробку програмного рішення.

4.1.7 Висновки щодо архітектури брокеру

В результаті розробки брокеру було створено просту архітектуру, яку легко модифікувати і змінювати. Для її реалізації використовується бібліотека Akka, що значно полегшує роботу з потоками і зменшує кількість багів, пов’язаних зі станом гонки та гонок даних.

Брокер складається з 5 частин, які відповідають за:

— адміністрування

— роботу черг

— зберігання даних у персистентному сховищі

— роботу з сесіями

— роботу з вводом-виводом

Компоненти поєднані між собою за допомогою інтерфейсів, що значно знижує зв’язність і підвищує згуртованість коду.

4.2 Архітектура клієнта

4.2.1 Загальний огляд архітектури клієнта

Архітектура клієнту має багато спільного з архітектурою брокеру, але має і ряд відмінностей. Як і архітектура брокеру, на клієнті протокол має два рівні: логічний і транспортний. Транспортний рівень слідкує за гарантією передачі і працює з вводом-виводом. Логічний рівень слідкує за станом сесії, RTT та іншим. Схему архітектуру клієнта зображено на кресленику ІТ61.110БАК.004Д4

Клієнт складається з 6 концептуальних модулів: Адаптер API, серіалізатор, десереалізатор, модуль шифрування, менеджер сесії та мультиплексор вводу виводу. Ці модулі формують 4 незалежних шари клієнту:

— шар доступу ─ складається з адаптеру API. Це фасад для роботи з АПІ, який відрізняється для консьюмера і продюсера. Він виступає у якості проксі для подальших шарів;

— шар серіалізації ─ складається з серіалізатору та десеріалізатору. Відповідає за перетворення програмних об'єктів у формат, придатний до транспортування мережею та зворотні перетворення. Впроваджує інтерфейс, реалізувавши який можна використовувати клієнтську серіалізацію;

— шар шифрування ─ складається з модулю шифрування. На основі ключів, отриманих від шару додатку шифрує та дешифрує тіло повідомлення, що надходять з транспортного рівня або рівня серіалізації. Оскільки використовує алгоритм симетричного шифрування, то той самий модуль можна використовувати як у консьюмері, так і у продюсері.

— Транспортний шар ─ складається з менеджеру сесії та мультиплексору вводу-виводу. Менеджер сесії відповідає за підтримку з'єднання з брокером, а мультиплексор представляє з собою комбінацію UDP клієнту та серверу, що дозволяє виконувати дуплексну комунікацію між брокером та клієнтом.

Загалом, така архітектура є досить гнучкою і легкою для модифікації. Кожен шар та компонент має лише один обов'язок та причину для зміни, що задовільняє принципу єдиної відповідальності. Поділення шару серіалізації на 2 компоненти дозволяє задовільнити принципу сегрегації інтерфейсів, що дає змогу зробити додаток більш модульним.

Схематично архітектуру клієнту зображено на кресленику ІТ61.060БАК.004 Д4

4.2.2 Шифрування

Для шифрування використовується сучасний симетричний блочний алгоритм AES з довжиною ключа 256 біт. Система використовує скрізне шифрування, тому дані захищені у транзиті та зберіганні і дешифруються лише на стороні отримувача. Це дозволяє значно знизити ризики при знаходженні вразливостей у брокері. Для розподілення ключів використовується протокол Діффі-Хелмана. Для його забезпечення реалізоване двофакторне рукостискання: у поточній версії протоколу два простих числа константні. При запиті аутентифікації клієнт передає своє випадкове число. При відповіді сервер використовує це випадкове число і шифрує ключ сесії за допомогою обрахованого ключа, після чого у тілі відповіді передає своє випадкове число і зашифрований закритий ключ. Клієнт розшифровує повідомлення і отримує доступ до ключа шифрування.

Такий алгоритм має ряд недоліків, а саме статичність ключа і зберігання його на сервері, хоч до нього і складно отримати доступ. Цей алгоритм варто поліпшити у наступних версіях протоколу.

4.2.3 Серіалізація та десеріалізація

Для серіалізації та десеріалізації за замовченням використовується формат JSON. JSON — зручний для людини формат, у яком дані представляються як невпорядкований сет пар ключ-значення. Він дуже популярний і його головною перевагою є простота та зручність зневадження. З іншого боку використання такого формату має ряд недоліків:

дані у JSON представлені у неоптимальному вигляді. Не має можливості додавати свої типи даних;

JSON — неструктурований формат, тому його парсинг — не дешева операція;

окрім самих значень необхідно передавати ключі. Це значно збільшує розмір повідомлення, іноді на 60-80 відсотків.

Через це було прийнято рішення надати змогу користувачам впроваджувати свої механізми серіалізації та десеріалізації. Для цього необхідно реалізувати інтерфейс серіалізатору та десеріалізатору.

Для наступної версії протоколу бажано створити серіалізатори і для інших форматів. Формат XML хоч і втрачає популярність, але і по сьогоднішній день значно поширений у корпоративному секторі. Створення адаптеру для цього протоколу дозволить полегшити інтеграцію протоколу у вже існуючи додатки у цьому секторі.

На великих об’ємах даних використання JSON та XML є небажаним, тому створення адаптеру для роботи зі структурованими протоколами, такими як Protocol Buffers та Apache Thrift дозволить ще більше підкреслити основну перевагу протоколу — високу пропускну здатність.

4.2.4 Транспортний шар

Транспортний шар відповідає за механізми доставки. Він складається з UDP серверу та клієнту. Мультиплексор вводу-виводу базується на функціях зворотнього виклику і не гарантує доставки повідомлень сам по собі, він може лише нотифікувати зареєстровані функції зворотнього виклику про підтвердження доставки. Функції зворотнього виклику працюють на рівні сесій. Сесія має дві структури даних: чергу з пріоритетом для зберігання повідомлень, що дозволяє прискорити доставку важливих повідомлень. Окрім цього існує черга для повідомлень у транзиті, що яка використовується для зберігання повідомлень, які були відправлені, але не отримали підтвердження.

Для реалізації черги з пріоритетами використовується максимальна куча. Ця структура даних надає доступ до найпріоритетніших даних з високою швидкістю.

4.2.5 Особливості продюсера

Продюсер — частина системи, що генерує потік даних. Це важлива частина системи, оскільки продюсери бувають дуже різні — від мікросервісу, що знаходиться у датацентрі, до датчику, що розташований у незахищеній зовнішній середі і під'єднаний до мережі через бездротові технологій.

Для передачі повідомлень використовується асинхронний API, тому результатом кожної операції є ComplitableFeature — сучасна абстракція асинхронного обчислення у світі Java.

Продюсер не має механізму backpressure і просто надсилає повідомлення до брокеру. У наступних версіях протоколу потрібно врахувати можливість відсилати декілька пакетів з вимогою доставки хоча б одного. Це актуально у ненадійних мережах, особливо у інтернеті речей.

4.2.6 Особливості консьюмера

Консьюмер — частина додатку, що відповідає за обробку повідомлень. Вона отримує повідомлення і обробляє їх. Для клієнтського API використовується event-driven модель. Це дозволяю створити простий і зрозумілий інтерфейс, завдяки якому програмісти реєструють функції зворотнього виклику і реагують на нові повідомлення. Програміст може обирати коли відправляти підтвердження обробки повідомлення: на початку або на після обробки повідомлення. Якщо підтвердження відправляється на початку, то отриманий режим роботи буде "не більше одного разу".

Головною проблемою консьюмера є те, що дуже часто саме він являється вузьким місцем системи. При тестуванні виявилось, що недостатня швидкість консьюмера призводить до значних проблем з швидкістю роботи брокера, тому необхідно продумати реалізацію backpressure механізму при розробці наступної версії протоколу.

4.2.7 Висновки щодо архітектури клієнту

При розробці клієнта було враховано необхідність легкості модифікації для подальшого розвитку протоколу. Було враховано необхідність зручного API для роботи з протоколом. Для цього було реалізовано два різних варіанти: для консьюмера і для продюсера.

API продюсера дає змогу використовувати сучасний асинхронний інтерфейс на основі CompletableFeature. Його використання інтуїтивно зрозуміло. У подальших версіях протоколу бажано врахувати можливість декількох режимів відправки, а не лише гарантований режим.

API консьюмера базується на event-driven підході і реалізує API на основі патерну функцій зворотнього виклику(більш елегантна версія патерну Observer).

Головною проблемою при створені клієнту стала відсутність backpressure можливостей у протоколі. При значній різниці у швидкості генерації та обробки повідомлень черга переповнюється, що призводить до блокувань і значного зниження швидкодії додатку. У наступній версії протоколу необхідно або змінити модель самого протоколу, або придумати механізм backpressure, що буде повідомляти продюсерів про надлишкову швидкість генерації повідомлень і тим самим знизить навантаження на брокер та консьюмерів.

Висновки до розділу

При реалізації протоколу, було виявлено перелік проблем, розв’язання яких є критичним для подальшого розвитку протоколу.

Головною проблемою стала відсутність зворотнього зв’язку між консьюмерами та продюсерами — якщо продюсери генерують повідомлення занадто швидко, то консьюмери та брокер перевантажуються, що викликає втрату пакетів через блокування і призводить до від’єднання продюсерів від

брокеру. Це головна проблема, яку дуже важливо розв’язати у наступній версії протоколу.

Протокол впроваджує мінімальне необхідне API для використання у реальних системах. Він впроваджує можливості гарантованої доставки і механізми захисту даних у транзиті. Але залишається проблема використання статичного ключа для шифрування. Механізм зміни ключа через певний проміжок часу міг би розв’язати цю проблему, але це вимагає переведення протоколу у дуплексний режим на логічному рівні: відтепер сервер буде надсилати не лише підтвердження про доставку, а й інші команди до продюсерів та консьюмерів.

Окрім протоколу було розроблено брокер та два варіанту клієнта: консьюмер та продюсер. Для реалізації було використано мову програмування Java. При розробці додатків було враховано потреби гнучкості та легкості модифікації коду. Вихідний код має високі показники згуртованості і низькі показники зв’язності, що дозволить значно легше змінювати модулі у середині та підміняти їх реалізації.

При розробці модулів було враховано принципи SOLID та DRY. При створенні клієнту було враховано сучасні тенденції у дизайні API та створено прості у використанні інтерфейси.

Важливою властивістю протоколу стала дуплексність під’єднань – кожний екземпляр під’єднання виконує роль як клієнту так і серверу. У зв’язку з цим у вапабдках брокеру та прод’юсеру необхідно використовувати як мінімум два потоки – один для очікування вводу-виводу з транспортного шару і один для виконання інших обчислень. Також у подальшому у протоколі можна розвивати можливості комунікації точка-точка.

При розробці брокеру було приділено увагу багатопоточності та можливим станам гонки. Також було звернуто увагу на сегрегацію маршрутизацій повідомлень між мережевими під’єднаннями та чергами і партиціями.

При розробці API продюсера було використано актуальний підхід до асинхронних обчислень і обрано абстракцію Completable Future. Це дозволяє легко композувати подальші обчислення і будувати ланцюги асинхронних операцій.

При розробці API консьюмера було обрано шаблон проєктування Observer(спостерігач) та його реалізацію через функцію зворотнього виклику. Це дозволяє значно зменшити кількість структурного коду, який ускладнює розуміння логіки програми.

5 ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

5.1 Юніт тестування

Юніт тестування, також відоме як модульне тестування — тестування компонентів програми у ізольованому середовищі. Юніт тестування дозволяє перевірити коректність роботи окремих компонентів в ізоляції від інших компонентів та зовнішнього світу. За допомогою юніт тестів можна створити умови, в яких компонент буде вести себе детерміновано і не залежатиме від сайд ефектів.

Юніт тестування використовується для перевірки працездатності сесійного рівня протоколу брокеру та клієнту, серіалізатору та десеріалізатору, логіки обробки пакетів і парсингу пакетів з масиву байт.

5.2 Інтеграційне тестування

Інтеграційне тестування використовується для перевірки коректності взаємодії модулів. На відміну від юніт тестів при інтеграційних тестах відбувається взаємодія з зовнішнім середовище, тому необхідно мінімізувати кількість таких тестів.

У проєкті інтеграційне тестування використовується для тестування взаємодії продюсера та брокеру, брокера та консьюмера та встановлення сесії на транспортному рівні.

5.3 Мануальне тестування

Мануальне тестування виконується людиною. При цьому перевіряється працездатність додатку у ручному режимі. Для доведення працездатності додатку було створено невеликий демо-додаток, який демострує відправку пакетів продюсером та їх отримання консьюмером. Результат роботи програми можна переглянути у режимі реального часу. Демо програма представляє

собою два консольні додатки: консоль продюсера та консоль консьюмера. У консоль продюсера можна писати повідомлення, при цьому ці повідомлення передаються мережею і виводяться у другу консоль.

5.4 Навантажувальне тестування

Навантажувальне тестування дозволяє перевірити швидкодію додатку. При тестуванні поточної версії протоколу було проведено тестування на 100, 500, 1000, 5000 та 10000 повідомлень в секунду при рівномірному споживані та надходженню повідомлень до черги. Для тестування використовується одна черга з однією партицією.

В результаті тестування було виявлено, що при 100, 500 та 1000 повідомлень у секунду черга працює без значних затримок. При 5000 та 10000 повідомлень на секунду з’являються досить великі однократні затримки. Це пов’язано зі збіркою мусору на стороні брокеру та клієнту. Оптимізація та зменшення кількості аллокацій на хіпі може значно покращити результати. Також можливе покращення результату при використані мов без механізму збірки сміття, таких як C++, Rust або D.

Висновки до розділу

Для забезпечення належної якості програмного забезпечення було проведено його тестування декількома способами. Для забезпечення коректності роботи окремих модулів використовуються юніт тести, для перевірки коректності взаємодії між модулями використовуються інтеграційні тести. Для перевірки пропускної здатності додатку використовується навантажувальне тестування. Також існує невелике демо, у якому можна відлагоджувати проект у ручному режимі.

ВИСНОВКИ

В ході виконання дипломного проєкту було розроблено комплекс програмного забезпечення, який реалізує систему комунікації за моделлю продюсер-консьюмер.

Було обґрунтовано актуальність, наукову та практичну цінність подібного рішення. Було складено технічні вимоги до рішення. Для цього було проаналізовано існуючи рішення, такі як RabbitMQ та Kafka.

Було розглянуто технології, які значно спростили розробку додатку та задовільнити висунуті вимоги до ПЗ. Для забезпечення кросплатформності у якості платформи виконання було обрано віртуальну машину Java та мову програмування Java. Для забезпечення високої пропускної здатності було обрано асинхронні методи вводу-виводу. Для створення панелі керування було обрано фреймворк для розробки веб-застосунків Spring

При розробці специфікації протоколу було враховано необхідність гарантованої доставки повідомлень. Для цього було розроблено декілька алгоритмів. Для гарантії доставки використовується алгоритм, схожий з протоколом QUIC — на кожне повідомлення відправляється підтвердження. Для контролю заторів використовується алгоритм повторної відправки повідомлень з експоненційною затримкою між спробами.

Для забезпечення конфіденційності даних у транзиті протокол використовує скрізне шифрування з шифром AES і довжиною ключа 256 біт. Ключі поширюються за допомогою алгоритму Діфі-Хелману.

При розробці брокеру та клієнту було враховано можливість повторно використовувати код транспортного шару. При проєктуванні компонентів було враховано принципи DRY та SOLID. Це дозволило зробити код компонентів легким у підтримці і надати великі можливості для подальшого розвитку функціоналу.

Було проведено тестування компонентів системи для верифікації працездатності програмного комплексу. Для тестування компонентів в ізоляції

від зовнішнього середовища було використано юніт тестування. Для перевірки коректності взаємодії компонентів було використано інтеграційне тестування. Для перевірки пропускної здатності системи було використано навантажувальне тестування. Також було створено невеликий демо проєкт, що дозволяє у інтерактивному режимі перевірити роботу системи.

Також було виявлено певні недоліки у отриманому рішенні. Одним з основних недоліків є відсутність режиму негарантованої доставки та обмеження на одного читача з партиції. При режим негарантованої доставки є дуже актуальним у мережах з великим об’ємами передаваних даних та низькою надійністю мережі. Присутні проблеми з шифруванням. Варто створити режим роботи без шифрування та шифрування з використанням асиметричних алгоритмів. Також у поточній версії було виявлено проблеми зі станом гонки при великих навантаження, які можна розв’язати використавши функціональну парадигму програмування та іммутабельні структури даних. Також існує проблема зниження продуктивності при великих навантаженнях через потребу збору великої кількості невикористовуваних об’єктів. Цю проблему можна розв’язати оптимізувавши аллокації пам’яті на кучі та ручним налаштуванням збірника сміття.

В цілому під час виконання проєкту було виконано всі поставлені задачі та реалізовано працездатний протокол для асинхроного обміну повідомленнями. Було виявлено недоліки, які необхідно врахувати при подальшому розвитку протоколу.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Hypertext transfer protocol version 3 (HTTP/3) [Електронний ресурс] : Запропонований стандарт / IETF ; M. Bishop, Ed. Akamai. - 5 с. - Доступ : https://datatracker.ietf.org/doc/draft-ietf-quic-http/

2. Kafka: the defenitive guide : Довідник / N. Narkhede, G. Shapira, та T. Palino. - M.: O'Reilly, 2017. - 10 с.

3. RabbitMQ in Depth : Довідник / G. Roy - M.: Manning, 2018. - 3 с.

4. RFC 793. Transmission Control Protocol. Philosophy.

5. Functional Programming in Java : Довідник / P. Saumont - M.: Manning, 2016. - 15 с.

6. RFC 768. User Datagram Protocol. Introduction.

7. Dependency Injection: With Examples in Java, Ruby, and C# : Довідник / D. Prasanna - M.: Manning, 2015. - 21 с.

8. The C10K problem [Електронний ресурс] : Стаття / Kegel ; - Доступ : http://www.kegel.com/c10k.html

9. Introduction to non-blocking I/O [Електронний ресурс] : стаття / D. Kegel - Доступ : http://216.92.86.126/dkftpbench/nonblocking.html

10. The Node.js Event Loop, Timers, and process.nextTick() [Електронний ресурс] : Документація / Доступ : https://nodejs.org/ua/docs/guides/event-loop-timers-and-nexttick/

11. Go's work-stealing scheduler [Електронний ресурс] : Стаття / Доступ https://rakyll.org/scheduler/

12. Scala: Guide for Data Science Professionals : Довідник / P. Bugnion, A. Manivannan, P. Nicolas - M.: Packt, 2015. - 217 с.

13. Gradle in Action : Довідник / B. Muchko - M.: Manning, 2014. - 13 с.

14. "Understanding and Mitigating Packet Corruption in Data Center Networks" : University of Washington / D. Zhuo, M.Ghobadi, R. Mahajan та ін. - Доступ : https://people.csail.mit.edu/ghobadi/papers/corropt_sigcomm_2017.pdf

15. Green Information Technology 1st edition : Підручник / W. Mansouri, K. Ben Ali, M. Obaidat – 767 с.